RKNN SDK User Guide

文件标识：RK-KF-YF-417

发布版本：V2.4.0

日期：2026-01-17

文件密级：□绝密 □秘密 □内部资料 ■公开

免责声明

本文档按"现状"提供，瑞芯微电子股份有限公司（"本公司"，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

"Rockchip"、"瑞芯微"、"瑞芯"均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

版权所有 © 2026 瑞芯微电子股份有限公司

超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址： 福建省福州市铜盘路软件园A区18号

网址： www.rock-chips.com

客户服务电话： +86-4007-700-590

客户服务传真： +86-591-83951833

客户服务邮箱： fae@rock-chips.com

前言

概述

本文介绍Rockchip RKNN SDK的开发。

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

版本	修改人	修改日期	修改说明	核定人
V1.6.0	HPC	2023-11-28	初始版本	熊伟
V2.0.0-beta0	HPC	2024-03-15	1. 增加RK3576内容； 2. 增加稀疏化推理； 3. Matmul API接口改进，增加量化参数、动态shape输入、iommu_domain_id功能	熊伟
V2.1.0	HPC	2024-08-02	1. 增加RV1103B、RK2118内容； 2. 改进rknn_server说明； 3. 改进Matmul API接口，增加更多数据类型、layout类型、量化方式支持；	熊伟
V2.2.0	HPC	2024-09-04	1.修正部分图片和内容表述	熊伟
V2.3.0	HPC	2024-11-06	1.增加通过 pip 源安装 RKNN-Toolkit2 的支持 2.增加 RKNN-Toolkit2 的 ARM64 版本支持	熊伟
V2.3.2	HPC	2025-04-03	增加 RV1126B 平台支持	熊伟
V2.4.0	HPC	2026-01-17	更新版本号	熊伟

目录

RKNN SDK User Guide

- 1 RKNN简介
  - 1.1 RKNN工具链介绍
    - 1.1.1 RKNN软件栈整体介绍
    - 1.1.2 RKNN-Toolkit2功能介绍
    - 1.1.3 RKNN Runtime功能介绍
  - 1.2 RKNN开发流程介绍
  - 1.3 适用的硬件平台
  - 1.4 关键字说明
- 2 开发环境准备
  - 2.1 RKNN-Toolkit2安装
    - 2.1.1 通过Docker方式安装
      - 2.1.1.1 安装Docker工具
      - 2.1.1.2 镜像准备
      - 2.1.1.3 查询镜像信息
      - 2.1.1.4 运行镜像
      - 2.1.1.5 运行Demo
    - 2.1.2 通过Pip方式安装
      - 2.1.2.1 安装Python环境
      - 2.1.2.2 安装Miniforge工具
      - 2.1.2.3 创建RKNN-Toolkit2 Conda环境
      - 2.1.2.4 安装 RKNN-Toolkit2
  - 2.2 设备端NPU环境准备
    - 2.2.1 NPU驱动版本确认
    - 2.2.2 NPU连板环境确认
    - 2.2.3 RKNN Server安装和更新
      - 2.2.3.1 RK356X / RK3576 / RK3588 / RV1126B 平台
      - 2.2.3.2 RV1103 / RV1103B / RV1106 / RV1106B平台
    - 2.2.4 查看RKNN Server详细日志
      - 2.2.4.1 Android系统
      - 2.2.4.2 Linux系统
- 3 RKNN使用说明
  - 3.1 模型转换
    - 3.1.1 RKNN初始化及对象释放
    - 3.1.2 模型转换配置
    - 3.1.3 模型加载接口介绍
    - 3.1.4 构建RKNN模型
    - 3.1.5 导出RKNN模型
    - 3.1.6 模型转换工具RKNN
    - 3.1.7 RKNN-Toolkit2模型量化功能
  - 3.2 模型评估
    - 3.2.1 模型推理
    - 3.2.2 模型精度分析
    - 3.2.3 模型性能评估
    - 3.2.4 模型内存评估
  - 3.3 板端C API推理
  - 3.4 板端Python API推理
    - 3.4.1 系统依赖说明
    - 3.4.2 工具安装
    - 3.4.3 基本使用流程

3.4.4 运行参考示例
3.4.5 RKNN-Toolkit Lite2 API详细说明
3.4.5.1 RKNNLite初始化及对象释放
3.4.5.2 加载RKNN模型
3.4.5.3 初始化运行时环境
3.4.5.4 模型推理
3.4.5.5 查询SDK版本
3.4.5.6 查询模型可运行平台
3.5 矩阵乘法接口
3.5.1 主要用途和特点
3.5.2 Matmul API使用流程
3.5.3 矩阵乘法高级用法
3.5.3.1 指定量化参数的矩阵乘法
3.5.3.2 动态shape输入的矩阵乘法
3.5.4 高性能的数据排列方式
3.5.4.1 矩阵规格限制
4 示例
4.1 MobileNet模型部署示例
4.1.1 模型转换
4.1.2 模型连板运行
4.1.3 模型评估
4.1.3.1 精度评估
4.1.3.2 耗时评估
4.1.3.3 内存评估
4.1.4 板端部署
4.2 YOLOv5模型部署示例
4.2.1 模型转换
4.2.2 模型连板运行
4.2.3 板端部署运行
5 RKNN进阶使用说明
5.1 数据排列格式
5.2 RKNN Runtime零拷贝调用
5.2.1 零拷贝介绍
5.2.2 C API零拷贝整体流程
5.2.3 C API零拷贝的用法
5.3 NPU多核配置
5.3.1 多核运行配置方法
5.3.2 查看多核运行效果
5.3.3 多核性能提升技巧
5.4 动态Shape
5.4.1 动态Shape功能介绍
5.4.2 RKNN SDK版本和平台要求
5.4.3 生成动态Shape的RKNN模型
5.4.4 C API部署
5.4.4.1 通用API
5.4.4.2 零拷贝API
5.5 自定义算子
5.5.1 自定义算子介绍
5.5.2 整体流程介绍
5.5.2.1 使用RKNN-Toolkit2注册自定义算子并导出RKNN模型
5.5.2.2 编写自定义算子的C代码实现，通过RKNN API加载注册并执行
5.5.2.3 使用RKNN-Toolkit2连板推理或精度分析
5.5.3 Python端处理

5.5.4 C API部署
5.5.4.1 初始化自定义算子结构体
5.5.4.1.1 init回调函数
5.5.4.1.2 prepare回调函数
5.5.4.1.3 compute回调函数
5.5.4.1.4 destroy回调函数
5.5.4.2 注册自定义算子
5.5.4.3 模型推理
5.5.4.4 连板精度分析
5.6 多Batch使用说明
5.6.1 多Batch原理
5.6.2 多Batch使用方式
5.6.3 多Batch输入输出设置
5.7 RK3588/RK3576 NPU SRAM使用说明
5.7.1 板端环境要求
5.7.1.1 内核环境要求
5.7.1.2 RKNN SDK版本要求
5.7.2 使用方法
5.7.3 调试方法
5.7.3.1 SRAM是否启用查询
5.7.3.2 SRAM使用情况查询
5.7.3.3 通过RKNN API查询SRAM大小
5.7.3.4 查看模型SRAM的占用情况
5.8 模型剪枝
5.9 模型加密
5.10 Cacheable内存一致性
5.10.1 Cacheable内存同步的方向
5.10.2 同步Cacheable内存
5.11 模型稀疏化推理
5.11.1 稀疏化原理
5.11.2 训练稀疏化模型
5.11.3 RKNN稀疏化推理使用方法
5.11.4 RKNN稀疏化推理限制
5.12 生成部署C代码
5.13 ONNX模型编辑
5.13.1 onnx_edit接口说明
5.13.2 onnx_edit变换公式说明
5.13.3 变换公式示例
6 量化说明
6.1 量化介绍
6.1.1 量化定义
6.1.2 量化计算原理
6.1.3 量化误差
6.1.4 线性对称量化和线性非对称量化
6.1.5 Per-Layer量化和Per-Channel量化
6.1.6 量化算法
6.2 量化配置
6.2.1 量化数据类型
6.2.2 量化算法建议
6.2.3 量化校正集建议
6.2.4 量化配置方法
6.3 混合量化
6.3.1 混合量化用法

6.3.2 混合量化使用流程
6.3.3 自动混合量化
6.4 量化感知训练
6.4.1 QAT简介
6.4.2 QAT原理
6.4.3 QAT使用依据
6.4.4 QAT实现简例及配置说明
6.4.5 QAT支持的算子
6.4.6 QAT模型中浮点算子的处理
6.4.7 QAT经验总结
7 精度排查
7.1 模拟器精度排查
7.1.1 模拟器FP16精度
7.1.2 模拟器量化精度
7.2 Runtime精度排查
7.2.1 连板精度
7.2.2 Runtime精度
8 性能优化
8.1 模型性能优化前期分析流程
8.1.1 环境条件与配置检查
8.1.2 部署过程耗时分析
8.2 模型性能分析
8.2.1 获取Profile信息
8.2.2 分析逐层耗时
8.2.3 分析CPU算子影响
8.2.4 分析NPU算子性能瓶颈
8.3 量化加速
8.4 图级别优化
8.4.1 非NPU OP通过图变换实现NPU化
8.4.2 利用硬件Fuse特性设计网络或图优化
8.4.3 算法等效变换或者子图单OP化
8.4.4 算子等效进行“同类项合并”、“提取公因式”
8.5 算子级别优化
8.5.1 面向DDR性能优化的OP尺寸设计（非强制）
8.5.2 高利用率模型算子的设计
8.5.3 子图融合的匹配
9 内存使用优化
9.1 模型运行时内存组成及分析方法介绍
9.1.1 RKNN模型运行时内存组成
9.1.2 模型内存分析方法
9.2 如何使用外部分配内存
9.2.1 输入输出内存外部分配
9.2.2 模型内存的外部分配
9.3 Internal内存复用
9.4 多线程复用上下文
9.5 多种分辨率模型共享相同权重
10 常见问题
10.1 NPU环境准备问题
10.2 工具安装问题
10.3 模型转换常用参数说明
10.4 模型加载问题
10.4.1 RKNN-Toolkit2支持的深度学习框架和对应版本
10.4.2 各框架的OP支持列表

10.4.3 ONNX模型转换常见问题
10.4.4 Pytorch模型转换常见问题
10.4.5 TensorFlow模型转换常见问题
10.5 模型量化问题
10.6 模型转换问题
10.7 模拟器推理及连板推理的说明
10.8 模型评估常见问题
10.9 C API使用常见问题
11 相关资源

1 RKNN简介

1.1 RKNN工具链介绍

1.1.1 RKNN软件栈整体介绍

RKNN软件栈可以帮助用户快速的将AI模型部署到Rockchip芯片。整体的框架如下：

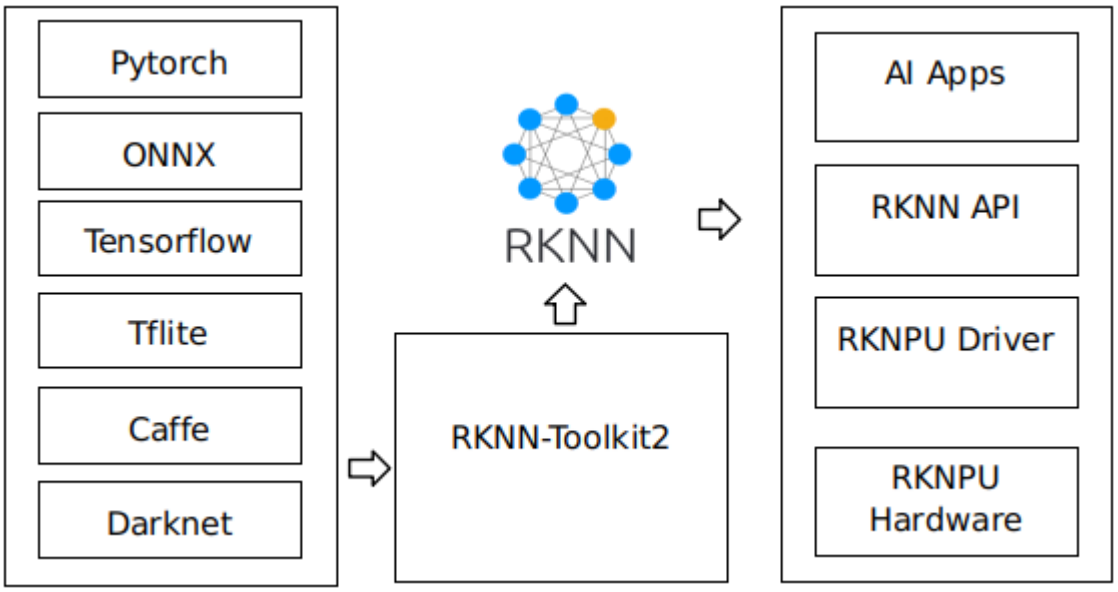


图 1-1 RKNN 软件栈

为了使用RKNPU，用户需要首先在计算机上运行RKNN-Toolkit2工具，将训练好的模型转换为RKNN格式模型，之后使用RKNN C API或Python API在开发板上进行部署。

1.1.2 RKNN-Toolkit2功能介绍

RKNN-Toolkit2是为用户提供在计算机上进行模型转换、推理和性能评估的开发套件，RKNN-Toolkit2的主要框图如下：

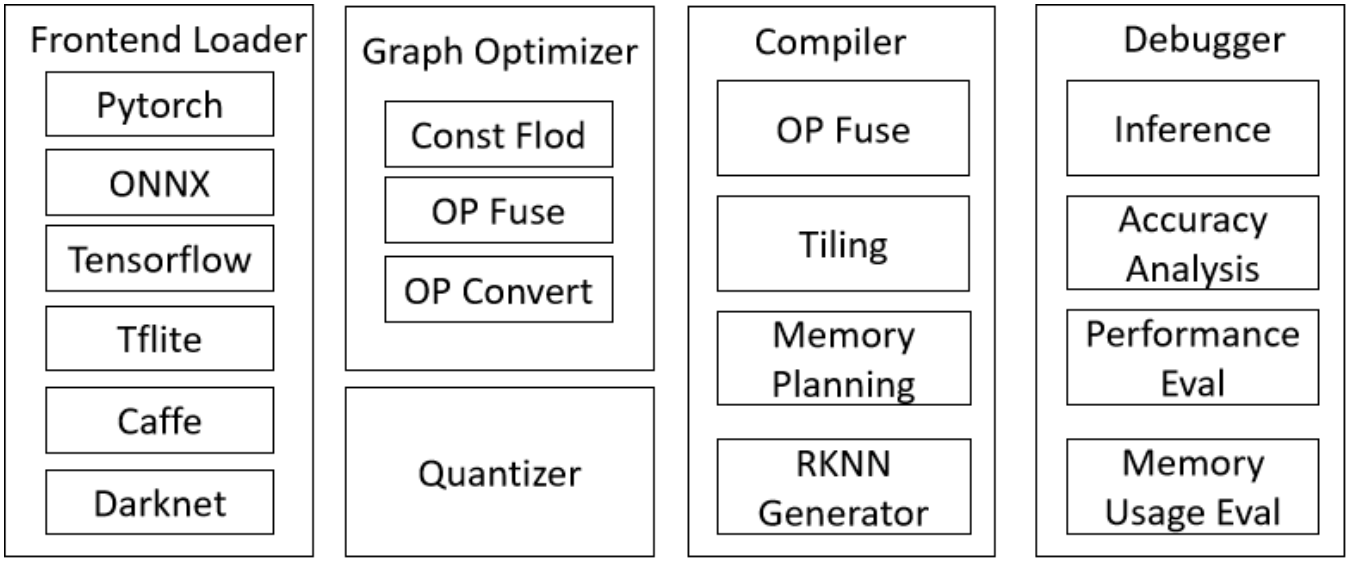


图 1-2 RKNN-Toolkit2 软件框图

通过该工具提供的Python接口可以便捷地完成以下功能：

1. 模型转换：支持将PyTorch、ONNX、TensorFlow、TensorFlow Lite、Caffe、DarkNet等模型转为RKNN模型。
2. 量化功能：支持将浮点模型量化为定点模型，并支持混合量化。
3. 模型推理：将RKNN模型分发到指定的NPU设备上进行推理并获取推理结果；或在计算机上仿真NPU运行RKNN模型并获取推理结果。
4. 性能和内存评估：将RKNN模型分发到指定NPU设备上运行，以评估模型在实际设备上运行时的性能和内存占用情况。
5. 量化精度分析：该功能将给出模型量化后每一层推理结果与浮点模型推理结果的余弦距离和欧氏距离，以便于分析量化误差是如何出现的，为提高量化模型的精度提供思路。
6. 模型加密功能：使用指定的加密等级将RKNN模型整体加密。

1.1.3 RKNN Runtime功能介绍

RKNN Runtime负责加载RKNN模型，并调用NPU驱动实现在NPU上推理RKNN模型。推理RKNN模型时，包括原始数据输入预处理、NPU运行模型、输出后处理三项流程。根据不同模型输入格式和量化方式，RKNN Runtime提供通用API和零拷贝API两种处理流程。

- 通用API推理：提供一套简洁、无门槛的推理API，易于使用，流程如图1-3所示。其中对数据的归一化、量化、数据排布格式转换、反量化等均在CPU上运行，模型本身的推理在NPU上运行。

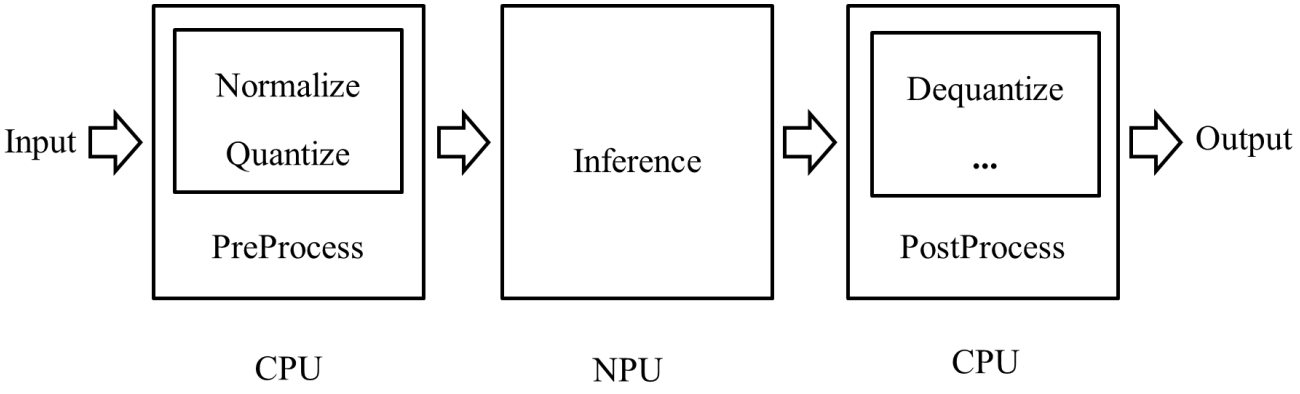


图1-3 通用API的数据处理流程

- 零拷贝API推理：流程如图1-4所示。优化了通用API的数据处理流程，归一化、量化和模型推理都会在NPU上运行，NPU输出的数据排布格式和反量化过程在CPU或者NPU上运行。零拷贝API对于输入数据流程的处理效率会比通用API高。支持数据在不同的IP核之间流动，没有数据拷贝，减少CPU及DDR带宽消耗。比如通过camera或者解码出来的数据，支持零拷贝导入到NPU中使用。

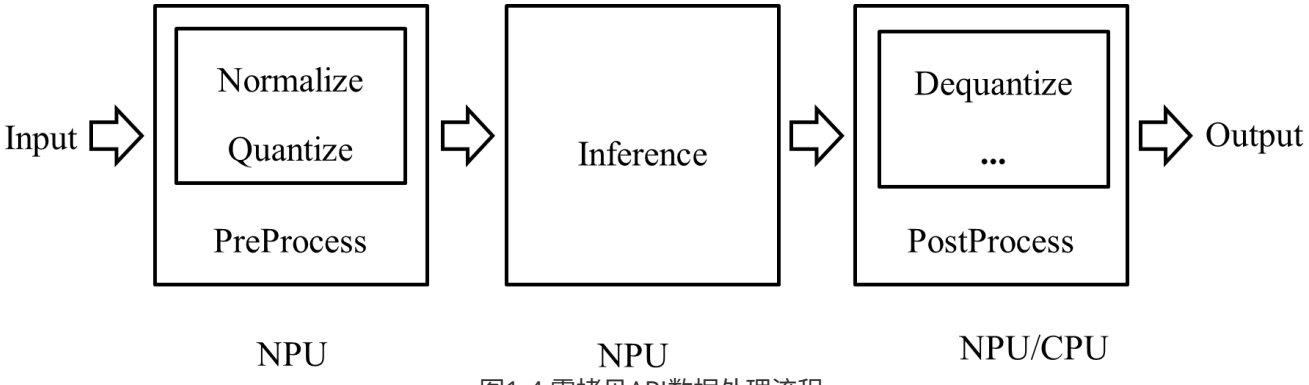


图1-4 零拷贝API数据处理流程

当用户输入数据只有虚拟地址时，只能使用通用API接口；当用户输入数据有物理地址或fd时，两组接口都可以使用。

1.2 RKNN开发流程介绍

用户可参考如下流程图了解RKNN的整体开发步骤，流程主要分为3个部分：模型转换、模型评估和板端部署运行。

图 1-5 RKNN 开发流程图

1. **模型转换**:

在这一阶段，原始的深度学习模型会被转化为RKNN格式，以便在RKNPU平台上进行高效的推理。这一阶段包括以下5个步骤：

a. 获取原始模型：获取或训练深度学习模型，建议使用主流框架，如ONNX、PyTorch或TensorFlow。

b. 模型配置：在RKNN-Toolkit2中进行必要的配置，如归一化参数、量化参数和目标平台等。

c. 模型加载：使用适当的加载接口将模型导入RKNN-Toolkit2，根据模型框架选择正确的加载接口。

d. 模型构建：通过`rknn.build()`接口构建RKNN模型，可选择是否进行量化，提高模型在硬件上推理的性能。

e. 模型导出：通过`rknn.export_rknn()`接口将RKNN模型导出成一个文件（.rknn 格式），用于后续部署。

具体内容请见3.1和4.2.1章节内容。

2.**模型评估**:

该阶段的主要目标是评估模型推理结果的准确性、推理性能和内存占用等关键指标。评估的主要步骤如下：

a. 模型连板调试：使用Python连接RKNPU平台对模型进行推理并验证结果准确性。这个阶段涵盖了输入数据的前处理和输出结果的后处理，以确保模型在板上运行正确。

b. 精度评估：通过`rknn.accuracy_analysis()`接口，比较量化模型与浮点模型推理结果之间的差异，以分析量化误差的来源。

c. 性能评估：通过`rknn.eval_perf()`接口，分析模型在RKNPU平台上的推理性能，帮助用户进一步优化模型结构，加快推理性能。

d. 内存评估：通过`rknn.eval_memory()`接口，了解模型在RKNPU平台上的内存使用情况，帮助用户进一步优化模型结构，最小化内存占用。

具体内容请参考3.2、3.3、4.2.2和4.2.3章节内容。

3. **板端部署运行**:

这个阶段涵盖了模型的实际部署和运行。它通常包括以下步骤：

a. 模型初始化：加载RKNN模型到RKNPU平台，准备进行前处理。

b. 模型前处理：加载待推理数据到RKNPU平台，准备进行推理。

c. 模型推理：执行推理操作，将输入数据传递给模型并获取推理结果。

d. 模型后处理：获取推理结果进行后处理，后处理结果传给应用端。

e. 模型释放：在完成推理流程后，释放模型资源，以便其他任务使用RKNN模型。

具体内容请参考3.4和4.2章节内容。

这三个步骤构成了完整的RKNN开发流程，确保人工智能模型能够成功转换、调试、评估和最终在RKNPU上高效部署。

1.3 适用的硬件平台

本文档适用如下硬件平台：

RK2118、RK3562、RK3566系列、RK3568系列、RK3576、RK3588系列、RV1103、RV1103B、RV1106、RV1106B、RV1126B

注：文档部分地方使用RK3566_RK3568来统一表示RK3566/RK3568系列，使用RK3588来统一表示RK3588/RK3588S系列。

1.4 关键字说明

RKNN模型：指运行在RKNPU上的模型文件，后缀名为.rknn。

连板推理：指通过USB口连接PC和开发板，调用RKNN-Toolkit2的接口运行模型。模型实际运行在开发板的NPU上。

DRM：英文全名Direct Rendering Manager，是一个主流的图形显示框架。

NATIVE_LAYOUT：指对于NPU运行时而言，通常性能表现最佳的计算机内存排列格式。

tensor：张量，在深度学习中，用它表示高阶数组的数据。

fd：文件描述符，被用来标识一块内存空间。

i8模型：量化的RKNN模型，即以8位有符号整型数据运行的模型。

fp16模型：非量化的RKNN模型，即以16位半精度浮点型数据运行的模型。

2 开发环境准备

RKNN-Toolkit2总共有支持两种不同的版本，一种是x86版本，另外一种是ARM64版本。两个版本的安装教程都一样，除了ARM64版本不支持docker安装。除此之外，ARM64版本仅支持转换模型时使用onnx和pytorch框架下的模型，其他框架的模型暂时不支持。

目前RKNN-Toolkit2的2个版本所支持深度学习框架的差异：

深度学习框架	x86版本	ARM64版本
Caffe	✔	✗
TensorFlow	✔	✗
TF Lite	✔	✗
ONNX	✔	✔
DarkNet	✔	✗
PyTorch	✔	✔

两者所依赖的工具差异：

依赖工具	x86版本	ARM64版本
adb	✔	✗
rknn_server	✔	✗
rknn runtime	✔	✔

两个版本使用差异：

主要差异在于调试功能包含精度分析，连板调试，内存和性能分析部分，ARM64版本直接在Linux系统的板端安装之后，只需直接使用上述对应功能，**不需要启动rknn_server**，只需保证板端上的`/usr/lib64`有对应的`librknnrt.so`，即可调用调试功能。由于ARM64版本不支持adb，所以所有调试功能都仅能在当前板端使用。x86版本需要更新并启动rknn_server，并且支持adb功能连接其他设备所以需保证adb正常使用状态下才能使用调试功能。

除此之外，x86版本和ARM64版本可能会存在性能差异，特别是在精度分析上或者使用大量图片做量化的情况下。

注：RK2118开发环境准备请参考《Rockchip_RK2118_Quick_Start_RKNN_SDK_CN》。

2.1 RKNN-Toolkit2安装

本章节提供两种RKNN-Toolkit2安装方式，通过Docker方式安装和通过pip方式安装，用户可自行选择任意一种方式进行安装。

2.1.1 通过Docker方式安装

2.1.1.1 安装Docker工具

已安装Docker工具的用户可跳过此步骤，未安装的用户请根据官方手册进行安装。Docker官方安装手册链接：https://docs.docker.com/install/linux/docker-ce/ubuntu/。

注意事项：需要将用户添加到docker用户组。

```
# 创建docker用户组
sudo groupadd docker

# 把当前用户加入docker用户组
sudo usermod -aG docker $USER

# 更新激活docker用户组
newgrp docker

# 验证不需要sudo执行docker命令
docker run hello-world
```

正确运行结果展示：

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:88ec0acaa3ec199d3b7eaf73588f4518c25f9d34f58ce9a0df68429c5af48e8d
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
```

2.1.1.2 镜像准备

本节介绍两种RKNN-Toolkit2镜像环境准备方式，可任选一种方式进行操作。

1. **通过Dockerfile创建镜像环境**

在RKNN-Toolkit2工程中docker/docker_file文件夹下，提供了构建RKNN-Toolkit2开发环境的Dockerfile文件，用户通过Docker命令创建镜像，如下所示。

```
# 注：以下 xx 和 x.x.x 代表版本号，请根据实际数值进行替换
cd docker/docker_file/ubuntu_xx_xx_cpxx
docker build -f Dockerfile_ubuntu_xx_xx_for_cpxx -t rknn-toolkit2:x.x.x-cpxx .
```

2. **加载已打包所有开发环境的Docker镜像**

下载对应版本的RKNN-Toolkit2工程文件，解压后在docker/docker_image文件夹下获取已打包所有开发环境的Docker镜像。

网盘下载链接：https://console.zbox.filez.com/l/I00fc3。提取码：rknn

执行以下命令加载对应Python版本的镜像文件。

```
# 注：x.x.x 代表 RKNN-Toolkit2 的版本号，cpxx 代表的是 Python 的版本号
docker load --input rknn-toolkit2-x.x.x-cpxx-docker.tar.gz
```

2.1.1.3 查询镜像信息

创建或加载镜像成功后，查看Docker的镜像信息。

```
docker images
```

相应的RKNN-Toolkit2镜像信息显示。

```
REPOSITORY      TAG         IMAGE ID        CREATED       SIZE
rknn-toolkit2   x.x.x-cpxx  xxxxxxxxxxxx    1 hours ago   5.89GB
```

2.1.1.4 运行镜像

执行以下命令运行Docker镜像，运行后将进入镜像的bash环境。

```
docker run -t -i --privileged -v /dev/bus/usb:/dev/bus/usb rknn-toolkit2:x.x.x-cpxx
/bin/bash
```

将文件夹examples代码映射进Docker环境可通过附加“-v <host src folder>:<image dst folder>”参数。

```
docker run -t -i --privileged -v /dev/bus/usb:/dev/bus/usb -v /your/rknn-toolkit2-
x.x.x/examples:/examples rknn-toolkit2:x.x.x-cpxx /bin/bash
```

2.1.1.5 运行Demo

```
cd examples/onnx/yolov5
python test.py
```

脚本运行成功后结果如下。

```
class   score  xmin, ymin, xmax, ymax
-------------------------------------------------
person  0.884 [ 208,  244,  286, 506]
person  0.868 [ 478,  236,  559, 528]
person  0.825 [ 110,  238,  230, 534]
person  0.339 [ 79,   353,  122, 516]
bus     0.705 [ 92,   128,  554, 467]
```

2.1.2 通过Pip方式安装

2.1.2.1 安装Python环境

若已安装Python环境，则可省略此步骤。

```
sudo apt-get update
sudo apt-get install python3 python3-dev python3-pip
sudo apt-get install libxslt1-dev zlib1g zlib1g-dev libglib2.0-0 libsm6 libgl1-mesa-glx
libprotobuf-dev gcc
```

若想安装RKNN-Toolkit2工具在本地环境（非Conda虚拟环境）中，请在安装好Python环境后跳至步骤2.1.2.4。

2.1.2.2 安装Miniforge工具

如果系统中同时有多个版本的Python环境，建议使用Miniforge管理Python环境。

检查是否安装Miniforge和版本信息，若已安装则可省略此小节步骤。

```
conda -V
# 提示 conda: command not found 则表示未安装miniforge conda
# 提示 例如版本 conda 23.9.0
```

下载Miniforge安装包

```
wget -c https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-Linux-
x86_64.sh
```

安装Miniforge

```
chmod 777 Miniforge3-Linux-x86_64.sh
bash Miniforge3-Linux-x86_64.sh
```

ARM64版本：

下载Miniforge安装包arm64版本

```
wget -c https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-Linux-
aarch64.sh
```

安装Miniforge arm64

```
chmod 777 Miniforge3-Linux-aarch64.sh
bash Miniforge3-Linux-aarch64.sh
```

2.1.2.3 创建RKNN-Toolkit2 Conda环境

进入Conda base环境

```
source ~/miniforge3/bin/activate  # Miniforge 安装的目录

# (base) xxx@xxx-pc:~\$
```

创建一个名为RKNN-Toolkit2且Python版本为3.8（建议版本）的Conda环境

```
conda create -n RKNN-Toolkit2 python=3.8
```

进入RKNN-Toolkit2 Conda环境

```
conda activate RKNN-Toolkit2
# (RKNN-Toolkit2) xxx@xxx-pc:~\$
```

2.1.2.4 安装 RKNN-Toolkit2

激活 RKNN-Toolkit2 Conda 环境后，可通过 pip 源或者本地 wheel 包这两种方式安装 RKNN-Toolkit2：

- 通过 pip 源安装

```
pip install rknn-toolkit2 -i https://pypi.org/simple

# 如果已安装 RKNN-Toolkit2，可通过以下命令升级 RKNN-Toolkit2
pip install rknn-toolkit2 -i https://pypi.org/simple --upgrade
```

- 通过本地 wheel 包安装

```
# 请根据不同的 python 版本及处理器架构，选择不同的 requirements 文件:
# 其中 cpxx 是 python 版本号
pip install -r packages/x86_64/requirements_cpxx.txt
# pip install -r packages/arm64/arm64_requirements_cpxx.txt

# 安装 RKNN-Toolkit2
# 请根据不同的 python 版本及处理器架构，选择不同的 wheel 安装包文件:
# 其中 x.x.x 是 RKNN-Toolkit2 版本号，cpxx 是 python 版本号
pip install packages/x86_64/rknn_toolkit2-x.x.x-cpxx-cpxx-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl
# pip install packages/arm64/rknn_toolkit2-x.x.x-cpxx-cpxx-
manylinux_2_17_aarch64.manylinux2014_aarch64.whl
```

若执行以下命令没有报错，则安装成功。

```
$ python3
>>> from rknn.api import RKNN
```

2.2 设备端NPU环境准备

2.2.1 NPU驱动版本确认

查询命令：

```
dmesg | grep -i rknpu
或
cat /sys/kernel/debug/rknpu/version
或
cat /sys/kernel/debug/rknpu/driver_version
或
cat /proc/debug/rknpu/driver_version
```

查询结果：

```
RKNPU driver: vX.X.X
```

X.X.X表示版本号，例如0.9.2。

Rockchip的固件均自带RKNPU驱动。若以上命令均查询不到NPU驱动版本，则可能为第三方固件未安装RKNPU驱动，需要打开kernel config文件的“CONFIG_ROCKCHIP_RKNPU=y”选项，重新编译内核驱动并烧录。建议RKNPU驱动版本>=0.9.2。

2.2.2 NPU连板环境确认

RKNN-Toolkit2的连板调试功能依赖板端的RKNN Server程序，该程序是一个运行在开发板上的后台代理服务，用于接收PC通过USB传输过来的命令和数据，然后调用相应的运行时接口，并返回相应结果给PC。所以在做连板调试前需要确认开发板是否已启动RKNN Server程序。

注：ARM64版本不需要通过连板调试，可跳过

检查方法：通过ADB工具进入开发板终端，查询是否有RKNN Server进程。

```
adb shell
ps | grep rknn_server
```

查询结果：

```
702 root 1192 S grep rknn_server
```

若查询到RKNN Server进程ID，则说明RKNN Server启动正常。若没有查询到RKNN Server进程ID，则执行以下命令手动启动RKNN Server后再进行查询。

Android系统手动启动RKNN Server：

```
su
setenforce 0
nohup /vendor/bin/rknn_server >/dev/null&
```

RK356X/ RK3576 / RK3588 / RV1126B 的Linux系统手动启动RKNN Server：

```
nohup /usr/bin/rknn_server >/dev/null&
```

RV1103 / RV1103B / RV1106 / RV1106B的Linux系统手动启动RKNN Server：

```
nohup /oem/usr/bin/rknn_server >/dev/null&
```

正常情况下Rockchip固件均已集成RKNN Server程序并添加到自启动脚本中，若无自启动或无相关脚本用于手动启动，请参考下一章节手动安装或更新RKNN Server。

2.2.3 RKNN Server安装和更新

RKNN-Toolkit2 的连板调试功能要求板端已安装 RKNPU2 环境，并且启动 RKNN Server 服务。以下是 RKNPU2 环境中的2个基本概念：

1. RKNN Server：一个运行在开发板上的后台代理服务，用于接收PC通过 USB 传输过来的数据，然后执行板端 Runtime 对应的接口，并返回结果给 PC。

2. RKNPU2 Runtime 库

- librknnrt.so: 是用于 RK3562 / RK3566 / RK3568 / RK3576 / RK3588 / RV1126B 板端的 Runtime 库。
- librknnmrt.so: 是用于 RV1103 / RV1103B / RV1106 / RV1106B板端的 Runtime 库。

如果板端没有安装 RKNN Server，Runtime 库不存在，或者 RKNN Server 和 Runtime 库的版本不一致，都需要重新安装或更新。

注意：

1. 若使用动态维度输入的 RKNN 模型，则要求 RKNN Server 和 RKNPU2 Runtime 库版本 >= 1.5.0。

2. 若使用大于2GB的模型，要求rknn_server版本>=2.0.0b0。

3. 在RV1103 / RV1103B / RV1106 / RV1106B等小内存平台上，建议rknn_server版本>=2.1.0。

4. 要保证 RKNN Server 、Runtime 库版本、RKNN-Toolkit2 版本一致，建议都安装最新的版本。

5. RK2118中的RKNN Server并不是用来做RKNN-Toolkit2的连板调试功能，而是用于DSP与MCU进行交互的服务。

6. ARM64版本无需依赖RKNN Server，所以不用启动和更新，但是调试功能依赖板端的RKNPU2 Runtime 库，确保板端有这个库并更新到最新版

2.2.3.1 RK356X / RK3576 / RK3588 / RV1126B 平台

1. Android系统

查询RKNN Server服务和`librknnrt.so`库版本，若版本号不一致请更新至同一版本。

```
# 查询rknn_server版本
strings /vendor/bin/rknn_server | grep -i "rknn_server version"
# 显示rknn_server版本为X.X.X
# rknn_server version: X.X.X

# 查询librknnrt.so库版本
# 64位系统
strings /vendor/lib64/librknnrt.so | grep -i "librknnrt version"
# 32位系统
strings /vendor/lib/librknnrt.so | grep -i "librknnrt version"
# 显示librknnrt库版本为X.X.X
# librknnrt version: X.X.X
```

更新RKNN Server服务和`librknnrt.so`库。

```
adb root
adb remount

# 64位系统:
adb push runtime/Android/rknn_server/arm64/rknn_server /vendor/bin/
adb push runtime/Android/librknn_api/arm64-v8a/librknnrt.so /vendor/lib64

# 32位系统:
adb push runtime/Android/rknn_server/arm/rknn_server /vendor/bin/
adb push runtime/Android/librknn_api/armeabi-v7a/librknnrt.so /vendor/lib

# 重启RKNN Server服务:
adb shell
su
chmod +x /vendor/bin/rknn_server
nohup /vendor/bin/rknn_server >/dev/null&
sync
reboot
```

2. Linux系统

查询RKNN Server服务和`librknnrt.so`库版本，若不一致请更新至同一版本。

```
# 查询rknn_server版本
strings /usr/bin/rknn_server | grep -i "rknn_server version"
# 显示rknn_server版本为X.X.X
# rknn_server version: X.X.X

# 查询librknnrt.so库版本
strings /usr/lib/librknnrt.so | grep -i "librknnrt version"
# 显示librknnrt库版本为X.X.X
# librknnrt version: X.X.X
```

更新RKNN Server服务和librknnrt.so库。

64位系统:

```
adb push runtime/Linux/rknn_server/aarch64/usr/bin/* /usr/bin
adb push runtime/Linux/librknn_api/aarch64/librknnrt.so /usr/lib64
```

32位系统：

```
adb push runtime/Linux/rknn_server/armhf/usr/bin/* /usr/bin
adb push runtime/Linux/librknn_api/armhf/librknnrt.so /usr/lib
```

重启RKNN Server服务：

```
adb shell
chmod +x /usr/bin/rknn_server
nohup /usr/bin/rknn_server >/dev/null&
restart_rknn.sh
```

2.2.3.2 RV1103 / RV1103B / RV1106 / RV1106B平台

查询RKNN Server和librknnmrt.so库版本，若不一致请更新至同一版本。

```
# 查询rknn_server版本
strings /oem/usr/bin/rknn_server | grep -i "rknn_server version"
# 显示rknn_server版本为X.X.X
# rknn_server version: X.X.X

# 查询librknnmrt.so库版本
strings /oem/usr/lib/librknnmrt.so | grep -i "librknnmrt version"
# 显示librknnmrt库版本为X.X.X
# librknnmrt version: X.X.X
```

更新RKNN Server服务和`librknnmrt.so`库。RV1103 / RV1103B 与RV1106 / RV1106B使用同一份RKNN Server服务和librknnmrt.so库。

```
adb push runtime/Linux/rknn_server/armhf-uclibc/usr/bin/* /oem/usr/bin
adb push runtime/Linux/librknn_api/armhf-uclibc/librknnmrt.so /oem/usr/lib

# 重启RKNN Server服务：
adb shell
chmod +x /oem/usr/bin/rknn_server
nohup /oem/usr/bin/rknn_server >/dev/null&
restart_rknn.sh
```

2.2.4 查看RKNN Server详细日志

2.2.4.1 Android系统

进入开发板终端，设置日志等级。

```
adb shell
su
setenforce 0
setprop persist.vendor.rknn.server.log.level 5
```

关闭当前RKNN Server服务进程。

```
kill -9 `pgrep rknn_server`
```

若没有自动重启RKNN Server服务，可以手动启动，查看详细日志。

```
nohup /vendor/bin/rknn_server >/dev/null&
logcat
```

2.2.4.2 Linux系统

进入开发板终端，设置日志等级。

```
adb shell
export RKNN_SERVER_LOGLEVEL=5
```

重启RKNN Server服务可查看详细日志。

针对RK356X / RK3576 / RK3588 / RV1126B 平台，执行下列命令。

```
nohup /usr/bin/rknn_server >/dev/null&
```

针对RV1103 / RV1103B / RV1106 / RV1106B平台，执行下列命令。

```
nohup /oem/usr/bin/rknn_server >/dev/null&
```

3 RKNN使用说明

3.1 模型转换

RKNN-Toolkit2提供了丰富的功能，包括模型转换、性能分析、部署调试等。本节将重点介绍RKNN-Toolkit2的模型转换功能。模型转换是RKNN-Toolkit2的核心功能之一，它允许用户将各种不同框架的深度学习模型转换为RKNN格式以在RKNPU上运行。用户可参考如下模型转换流程图以理解如何进行模型转换。

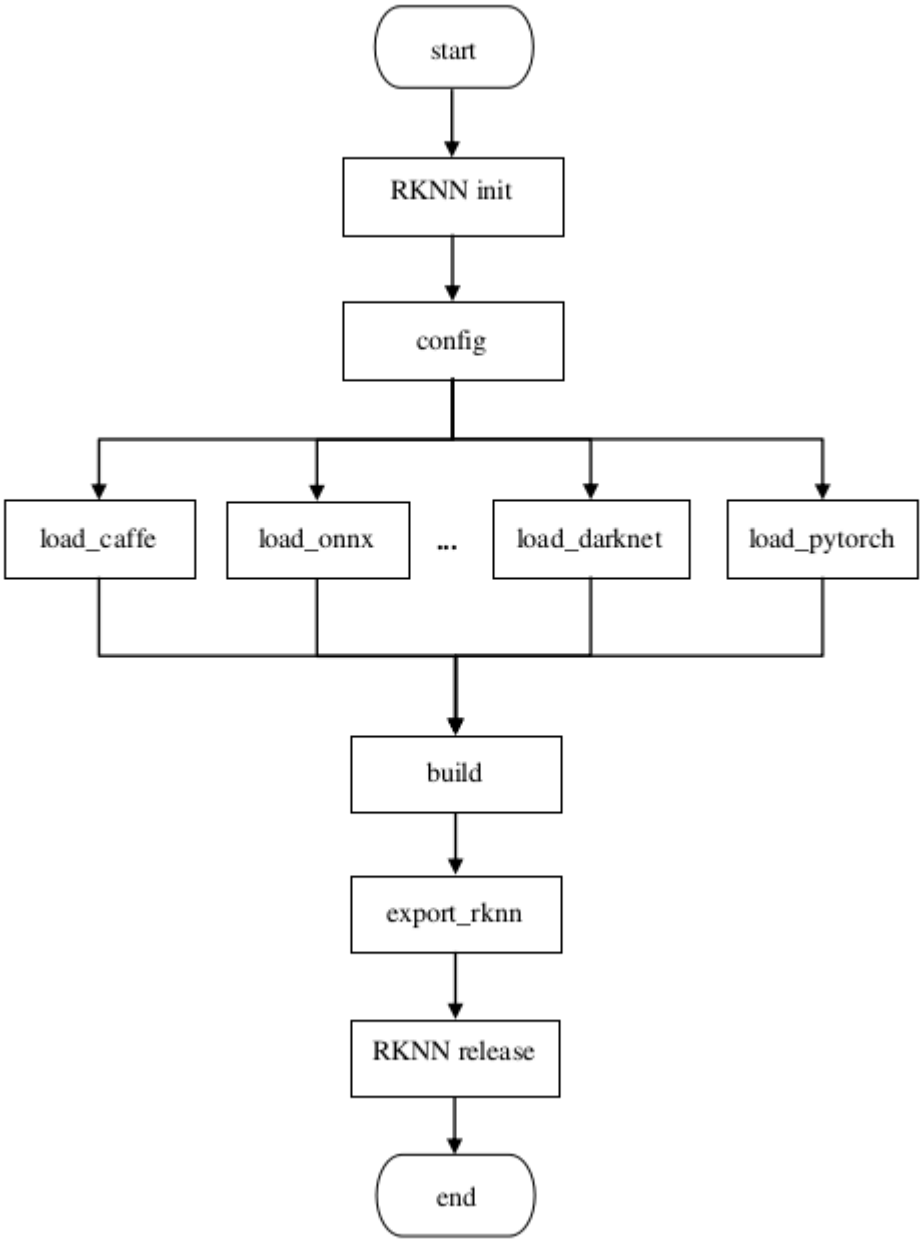


图3-1 模型转换流程图

目前RKNN-Toolkit2支持多个主流深度学习框架的模型转换，包括：

- Caffe（推荐版本为1.0）
- TensorFlow（推荐版本为1.12.0~2.8.0）
- TensorFlow Lite（推荐版本为Schema version = 3）
- ONNX（推荐版本为1.7.0~1.10.0）
- PyTorch（推荐版本为1.6.0~1.13.1）

- Darknet（推荐版本为Commit ID = 810d7f7）

用户可以使用上述框架训练或获取预训练模型并将它们转换为RKNN格式，方便更高效地在RKNPU平台上部署和推理。

3.1.1 RKNN初始化及对象释放

在这一阶段，用户需要先初始化RKNN对象，这是整个工作流程的第一步：

初始化RKNN对象：

- 使用`RKNN()`构造函数来初始化RKNN对象，用户可以传入参数`verbose`和`verbose_file`。
- `verbose`参数决定是否在屏幕上显示详细日志信息。
- 如果设置了`verbose_file`参数并且`verbose`为`True`，日志信息还将写入到指定的文件中。

示例代码：

```
rknn = RKNN(verbose=True, verbose_file='./mobilenet_build.log')
```

当完成所有的RKNN相关的操作后，用户需要释放资源，这是整个工作流程的最后一步：

- 使用`release()`接口释放RKNN对象占用的资源。

示例代码：

```
rknn.release()
```

3.1.2 模型转换配置

在模型转换之前，用户需要进行一些配置以确保模型转换的正确性和性能。配置参数通过`rknn.config()`接口设置，包括归一化参数、量化方法、目标平台等。下面列出了一些常用的配置参数：

- mean_values和std_values: 用于设置输入的均值和归一化值。这些值在量化过程中使用，且C API推理阶段图片不需再做均值和归一化值，减少部署耗时。
- quant_img_RGB2BGR: 用于控制量化阶段加载的量化校正图像是否需要进行RGB到BGR的转换，默认值为False。该配置只在量化时生效，模型部署阶段不会生效，需要用户在处理输入数据时提前做好相应转换。注：quant_img_RGB2BGR = True时输入数据的处理顺序为先做RGB2BGR转换（用户自行完成）再做归一化操作（运行时内部完成），详细注意事项请参考10.3章节。
- target_platform: 用于指定RKNN模型的目标平台，支持RK2118、RK3562、RK3566、RK3568、RK3576、RK3588、RV1103、RV1103B、RV1106、RV1106B、 RV1126B。
- quantized_algorithm: 用于指定计算每一层的量化参数时采用的量化算法，可以选择normal、mmse或kl_divergence，默认算法为normal，详细说明参考3.1.7、6.1和6.2章节。
- quantized_method: 支持`layer`或`channel`，用于每层的权重是否共享参数，默认为`channel`，详细说明见3.1.7、6.1和6.2章节。
- optimization_level: 通过修改模型优化等级，可以关掉部分或全部模型转换过程中使用到的优化规则。该参数的默认值为 3，打开所有优化选项，值为2或1时关闭一部分可能会对部分模型精度产生影响的优化选项，值为0时关闭所有优化选项。
- quantized_dtype: 用于指定量化类型，目前支持线性非对称量化的INT8，默认为'asymmetric_quantized-8'。

- custom_string: 添加自定义字符串信息到RKNN模型，例如模型自身迭代的版本信息等。在模型部署阶段可以通过`rknn_query`接口设置`RKNN_QUERY_CUSTOM_STRING`标志查询这些信息，方便部署时根据模型版本做特殊处理。默认值为`None`。
- dynamic_input: 支持动态输入，根据用户指定的多组输入shape，来模拟动态输入的功能，默认值为None，详细说明见5.4章节。

更具体的`rknn.config()`接口配置请参考`<Rockchip_RKNPU_API_Reference_RKNN_Toolkit2_CN>`手册。

示例代码：

```
rknn.config(
        mean_values=[[103.94, 116.78, 123.68]],
        std_values=[[58.82, 58.82, 58.82]],
        quant_img_RGB2BGR=False,
        target_platform='rk3566')
```

3.1.3 模型加载接口介绍

用户需要使用相应的加载接口导入不同框架模型文件。RKNN-Toolkit2提供了不同的加载接口，包括Caffe、TensorFlow、TensorFlow Lite、ONNX、PyTorch等，下面是各种框架的模型加载接口的简要介绍：

- Caffe模型加载接口：
  - 使用`rknn.load_caffe()`接口加载Caffe模型。
  - 需要提供模型文件（`.prototxt`后缀）路径和权重文件（`.caffemodel`后缀）路径。
  - 如果模型有多个输入层，可以指定输入层名称的顺序。

  示例代码：

  ```
  ret = rknn.load_caffe(model='./mobilenet_v2.prototxt',
  blobs='./mobilenet_v2.caffemodel')
  ```

- TensorFlow 模型加载接口：
  - 使用`rknn.load_tensorflow()`接口加载TensorFlow模型。
  - 需要提供TensorFlow模型文件（`.pb`后缀）路径、输入节点名、输入节点的形状以及输出节点名。

  示例代码：

  ```
  ret = rknn.load_tensorflow(tf_pb='./ssd_mobilenet_v1_coco_2017_11_17.pb', inputs=
  ['Preprocessor/sub'], outputs=['concat', 'concat_1'], input_size_list=[[1, 300, 300,
  3]])
  ```

- TensorFlow Lite 模型加载接口：
  - 使用`rknn.load_tflite()`接口加载TensorFlow Lite模型。
  - 需要提供TensorFlow Lite模型文件（.tflite后缀）路径。

  示例代码：

  ```
  ret = rknn.load_tflite(model='./mobilenet_v1.tflite')
  ```

- ONNX 模型加载接口：
  - 使用`rknn.load_onnx()`接口加载ONNX模型。
  - 需要提供 ONNX 模型文件（.onnx 后缀）路径。

  示例代码：

  ```
  ret = rknn.load_onnx(model='./arcface.onnx')
  ```

- DarkNet 模型加载接口：
  - 使用`rknn.load_darknet()`接口加载DarkNet模型。
  - 需要提供DarkNet模型文件（`.cfg`后缀）路径和权重文件（`.weights`后缀）路径。

  示例代码：

  ```
  ret = rknn.load_darknet(model='./yolov3-tiny.cfg', weight='./yolov3.weights')
  ```

- PyTorch 模型加载接口：
  - 使用`rknn.load_pytorch()`接口加载PyTorch模型。
  - 需要提供PyTorch模型文件（.pt后缀）路径，模型必须是torchscript格式的。

  示例代码：

  ```
  ret = rknn.load_pytorch(model='./resnet18.pt', input_size_list=[[1, 3, 224, 224]])
  ```

  用户可以根据不同框架的模型选择合适的接口进行加载，确保模型转换的正确性。

3.1.4 构建RKNN模型

用户加载原始模型后，下一步就是通过`rknn.build()`接口构建RKNN模型。构建模型时，用户可以选择是否进行量化，量化可以减小模型的大小和提高在RKNPU上的性能。`rknn.build()`接口参数如下：

- do_quantization: 该参数控制是否对模型进行量化，建议设置为`True`。
- dataset: 该参数指定用于量化校准的数据集，数据集的格式是文本文件。

  dataset.txt示例:

  ```
  ./imgs/ILSVRC2012_val_00000665.JPEG
  ./imgs/ILSVRC2012_val_00001123.JPEG
  ./imgs/ILSVRC2012_val_00001129.JPEG
  ./imgs/ILSVRC2012_val_00001284.JPEG
  ./imgs/ILSVRC2012_val_00003026.JPEG
  ./imgs/ILSVRC2012_val_00005276.JPEG
  ```

  示例代码：

```
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
```

3.1.5 导出RKNN模型

用户通过`rknn.build()`接口构建了 RKNN 模型后，可以通过`rknn.export_rknn()`接口导出RKNN模型文件（`.rknn`后缀），以便后续模型的部署。`rknn.export_rknn()`接口参数如下：

- export_path: 导出模型文件的路径。
- cpp_gen_cfg: 该参数决定是否在导出模型的同时生成 C++ 部署示例。

示例代码：

```
ret = rknn.export_rknn(export_path='./mobilenet_v1.rknn')
```

这些接口涵盖了RKNN-Toolkit2模型转换阶段，根据不同的需求和应用场景，用户可以选择不同的模型配置和量化算法进行自定义设置，方便后续进行部署。

3.1.6 模型转换工具RKNN

`rknn_convert`是RKNN-Toolkit2提供的一套常用模型转换工具，通过封装上述API接口，用户只需编辑模型对应的`yml`配置文件，就可以通过指令转换模型。以下是如何使用`rknn_convert`工具的示例命令以及支持的指令参数：

```
python -m rknn.api.rknn_convert -t rk3588 -i ./model_config.yml -o ./output_path
```

通过使用上述命令和参数，用户可以将模型转换为RKNN格式，并将转换后的模型保存到指定的输出路径。支持的指令参数说明如下：

- -i: 模型配置文件（.yml）路径。
- -o: 转换后模型输出路径。
- -t: target_platform，目标平台可以选择rv1103，rv1103b, rv1106，rv1106b，rv1126b, rk2118，rk3562，rk3566，rk3568，rk3576或rk3588。
- -e: (选填) 评估连板运行时model的耗时和内存占用，若开启请输入-e。注：一定要连接相应开发板并正确设置`target_platform`，否则会报错，当有多设备时可通过`-d`参数指定设备ID。
- -a: (选填)评估生成的rknn模型精度，开启模拟器精度评估请输入-a "xx1.jpg xx2.jpg"，若要开启连板精度评估请配合-d参数使用。
- -v: (选填)指定是否要在屏幕上打印详细日志信息，若开启打印模式请输入-v。
- -d: (选填)单个adb设备使用-d，多adb设备使用-d device_id，device_id通过adb devices查询。

下面是一个参考的yml配置文件(`object_detection.yml`)：

```
models:
   # model output name
   name: object_detection
   # Original model framework
   platform: onnx
   # Model input file path
   model_file_path: ./object_detection.onnx
   # Describe information such as input and output shapes
   subgraphs:
     # model input tensor shape
     input_size_list:
```

```
      - 1,3,512,512
    # input tensor name
    inputs:
      - data
    # output tensor name
    outputs:
      - conv6-1
      - conv6-2
      - conv6-3
  # quantification flag
  quantize: true
  # Quantify dataset file path (relative yml path)
  dataset: ./dataset.txt
  configs:
    quantized_dtype: asymmetric_quantized-8
    # rknn.config mean_values
    mean_values: [127.5,127.5,127.5]
    # rknn.config std_values
    std_values: [128.0,128.0,128.0]
    # rknn.config quant_img_RGB2BGR
    quant_img_RGB2BGR: false
    # rknn.config quantized_algorithm
    quantized_algorithm: normal
```

这个配置文件包括了模型的名称、原始模型使用的框架、模型文件路径、输入输出信息、是否进行量化等详细信息。用户可以根据模型的特定需求编辑相应的配置文件。

模型转换配置详见下表：

表3-1 rknn_convert模型转换配置参数说明

参数名	填写内容
-name	模型输出名称
-platform	原始模型使用的框架，支持tensorflow、tflite、caffe、onnx、pytorch、darknet
-model_file_path	原始模型文件路径，适用于单模型文件输入，例：tensorflow、tflite、onnx、pytorch
-quantize	是否开启量化
-dataset	量化dataset文件路径（相对yml配置文件路径），若要开启accuracy_analysis此项必填
-prototxt_file_path	platform为caffe时，模型的prototxt文件
-caffemodel_file_path	platform为caffe时，模型的caffemodel文件
-darknet_cfg_path	platform为darknet时，模型的cfg文件
-darknet_weights_path	platform为darknet时，模型的weight文件

参数名	填写内容
-subgraphs	描述输入输出shape等信息。除特定框架外，一般情况下该参数及附带的子参数可不写，使用模型默认值
----input_size_list(子参数)	输入tensor的shape
----inputs(子参数)	输入tensor的名称
----outputs(子参数)	输出tensor的名称
-configs	对应rknn.config()配置
----quantized_dtype(子参数)	量化类型，RKNN_toolkit2: 可填写 [asymmetric_quantized-8]，不输入用默认值
----mean_values(子参数)	输入的均值归一数，模型为单输入RGB如[123.675,116.28,103.53]，若为多输入如[[123,116,103],[255,255,255]]
----std_values(子参数)	输入的方差归一数，模型为单输入RGB如[58.395,58.295,58.391]，若为多输入如[[127,127,127],[255,255,255]]
----quant_img_RGB2BGR(子参数)	用于控制量化时加载量化校正图像时是否需要先进行RGB到 BGR 的转换，默认值是False
----quantized_algorithm(子参数)	量化算法，可选['normal', 'kl_divergence', 'mmse']，默认为 normal
----quantized_method(子参数)	量化方式，RKNN_toolkit2可选['layer', 'channel']，默认为channel
----optimization_level(子参数)	设置优化级别。默认为3，表示使用所有默认优化选项
----model_pruning(子参数)	修剪模型以减小模型大小，默认为false，开启为true
----quantize_weight(子参数)	当quantize参数为false时，通过量化一些权重来减小rknn模型的大小。默认为false，开启为true
----single_core_mode(子参数)	是否仅生成单核模型，可以减小RKNN模型的大小和内存消耗。默认值为False。目前对RK3588/RK3576生效。默认值为 False

3.1.7 RKNN-Toolkit2模型量化功能

RKNN-Toolkit2提供两种量化方式和三种量化算法，用户可以通过rknn.config()中的quantized_algorithm和quantized_method参数来选择。以下是这些量化算法和特征量化方式的介绍：

- 三种量化算法：
    1. Normal量化算法：通过计算模型中特征（feature）的浮点数最大值和最小值来确定量化范围的最大值和最小值。算法的特点是速度较快，但是遇到特征分布不均衡时效果较差，推荐量化数据量一般为20-100张左右。

2. MMSE量化算法：通过最小化浮点数与量化反量化后浮点数的均方误差损失确定量化范围的最大值和最小值，能够一定程度的缓解大异常值带来的量化精度丢失问题。由于采用暴力迭代的方式，速度较慢，但通常会比normal具有更高的精度，推荐量化数据量一般为20-50张左右，用户也可以根据量化时间长短对量化数据量进行适当增减。
3. KL-Divergence量化算法：将模型特征（feature）中浮点数和定点数抽象成两个分布，通过调整不同的阈值来更新浮点数和定点数的分布，并根据KL散度衡量两个分布的相似性来确定量化范围的最大值和最小值。所用时间会比normal多一些，但比mmse会少很多，在某些场景下（feature分布不均匀时）可以得到较好的改善效果，推荐量化数据量一般为20-100张左右。

- 两种量化方式：
  1. Layer量化方式：Layer量化方式将同一层网络的所有通道作为一个整体进行量化，所有通道共享相同的量化参数。
  2. Channel量化方式：Channel量化方式将同一层网络的各个通道独立进行量化，每个通道有自己的量化参数。通常情况下，Channel量化方式比Layer量化方式具有更高的精度。
  3. 根据实际的模型量化效果和需求，用户可以选择合适的量化算法和特征量化方式，更详细的量化说明和原理请见第6章。

3.2 模型评估

3.2.1 模型推理

原始模型转换为RKNN模型后，可在模拟器或开发板上进行推理，对推理结果进行后处理检验其是否正确。若推理结果不正确，可以对量化模型进行精度分析和查看前后处理流程是否正确。模型推理阶段使用到的主要接口相关参数说明如下：

运行时初始化接口`rknn.init_runtime()`参数如下：

- target：目标硬件平台。默认值为`None`，推理在模拟器上进行。如果要获取板端实际推理结果，则该参数需要填入相应的平台（RK3562 / RK3566 / RK3568 / RK3588 / RK3576 / RV1103 / RV1103B / RV1106 / RV1106B / RV1126B）。
- device_id：设备编号。默认值为`None`。若有设置target则选择唯一一台设备进行推理。如果电脑连接多台设备连板推理时，需要指定填入相应的设备ID。若通过网络adb连接设备进行模型推理，则需要用户手动执行命令`adb connect [IP]`连接网络设备后再填入设备编号，通常为`[IP]`或`[IP:Port]`的形式。使用`adb devices`命令可以列出所有已连接设备。
- perf_debug：进行性能评估时是否开启`debug`模式。默认值为`False`，调用`rknn.val_perf()`接口可以获取模型运行的总时间。设为`True`时，可以获取到每一层的运行时间，RV1103、RV1103B、RV1106和RV1106B平台不支持。
- eval_mem：是否进入内存评估模式。默认值为`False`。设为`True`进入内存评估模式后，可以调用`rknn.eval_memory()`接口获取模型运行时的内存使用情况。

推理接口`rknn.inference()`参数如下：

- inputs：待推理的输入列表，格式为ndarray。
- data_format：输入数据的layout列表，只对4维的输入有效，格式为“`NCHW`”或“`NHWC`”。默认值为`None`，表示所有输入的layout都为`NHWC`。

示例代码：

```
ret = rknn.init_runtime(target=platform, device_id='515e9b401c060c0b')

# Preprocess
image_src = cv2.imread(IMG_PATH)
img = preprocess(image_src)

# Inference
outputs = rknn.inference(inputs=[img])

# Postprocess
outputs = postprocess(outputs)
```

注意事项：
1.在RV1103 / RV1106平台上推理时若遇到"`E RKNN: failed to allocate fd, ret: -1, errno: 12`"报错，可以在开发板终端运行`RkLunch-stop.sh`，关闭其他占用内存的应用后再连板推理。将`rknn_server`更新到2.1.0版本也可以缓解该问题。

2.在ARM64版本中`rknn.init_runtime()`无需添加device_id

3.2.2 模型精度分析

若量化模型推理结果有问题，可以使用`rknn.accuracy_analysis()`接口进行精度分析，查看每层算子的精度。

精度分析接口`rknn.accuracy_analysis()`参数如下：

- inputs：输入文件路径列表（格式包括`jpg`、`png`、`bmp`和`npy`）。
- output_dir：分析结果保存目录，默认值为'`./snapshot`'。
- target：目标硬件平台。默认值为`None`，使用仿真器进行精度分析。如果设置了target（RK3562 / RK3566 / RK3568 / RK3588 / RK3576 / RV1103 / RV1103B / RV1106 / RV1106B / RV1126B），则会获取运行时每一层的结果，并进行精度分析。
- device_id：设备编号。默认值为`None`。若有设置target则选择唯一一台设备进行精度分析。如果电脑连接多台设备时，需要指定相应的设备ID。若通过网络adb连接设备进行精度分析，则需要用户手动执行命令`adb connect [IP]`连接网络设备后再填入设备编号，通常为`[IP]`或`[IP]:[Port]`的形式。

注意事项：

1.精度分析时可能会因为模型太大且开发板上存储容量不够导致运行失败，可在开发板终端上使用`df -h`命令来确认存储容量，若空间不足请删除无用文件保证`output_dir`对应分区有足够的存储空间用于保存结果文件。

2.通过`rknn.load_rknn()`加载RKNN模型后，因RKNN模型缺失原始模型信息，因此无法使用模型精度分析功能。

3.在ARM64版本中`rknn.accuracy_analysis()`无需添加device_id，并且该版本不支持adb设备连接到其他设备做精度分析。

示例代码：

```
ret = rknn.accuracy_analysis(inputs=[img_path], target=platform,
device_id='515e9b401c060c0b')
```

精度分析结果如下图

```
# simulator_error: calculate the output error of each layer of the simulator (compared to the 'golden' value).
#              entire: output error of each layer between 'golden' and 'simulator', these errors will accumulate layer by layer.
#              single: single-layer output error between 'golden' and 'simulator', can better reflect the single-layer accuracy of the simulator.
# runtime_error: calculate the output error of each layer of the runtime.
#              entire: output error of each layer between 'golden' and 'runtime', these errors will accumulate layer by layer.
#              single_sim: single-layer output error between 'simulator' and 'runtime', can better reflect the single-layer accuracy of runtime.

layer_name                              simulator_error                          runtime_error
                                      entire             single                entire          single_sim
                                    cos      euc       cos      euc           cos      euc       cos      euc
-------------------------------------------------------------------------------------------------------------
[Input] images                    1.00000 | 0.0       1.00000 | 0.0
[exDataConvert] images_int8       1.00000 | 2.5780    1.00000 | 2.5780
[Conv] 128                        1.00000 | 2.5780    1.00000 | 2.5780       1.00000 | 2.5780    1.00000 | 0.0
[Conv] 286
[Relu] 131                        0.99977 | 37.674    0.99977 | 37.674       0.99977 | 37.682    1.00000 | 2.0193
[Conv] 132
[Relu] 133                        0.99949 | 53.789    0.99960 | 47.679       0.99949 | 53.790    1.00000 | 3.2487
....
[Sigmoid] 283_int8                0.99903 | 3.6972    0.99995 | 0.8721
[exDataConvert] 283               0.99903 | 3.6972    0.99996 | 0.7372       0.99908 | 3.5866    1.00000 | 0.1702
[Conv] 280
[Sigmoid] output_int8             0.99934 | 6.3987    0.99995 | 1.7267
[exDataConvert] output            0.99934 | 6.3987    0.99996 | 1.4713       0.99932 | 6.4315    1.00000 | 0.3788
```

图3-2 精度分析结果

完整的精度分析包括模拟器精度分析结果和板端精度分析结果，同时模拟器和板端的精度分析结果都分为完整模型推理结果对比和逐层推理结果对比。详细说明如下：

- simulator_error:
  - entire: 从输入层到当前层 simulator 结果与 golden 结果对比的余弦距离和欧氏距离。
  - single: 当前层使用 golden 输入时，simulator 结果与 golden 结果对比的余弦距离和欧氏距离。
- runtime_error:
  - entire: 从输入层到当前层板端实际推理结果与 golden 结果对比的余弦距离和欧氏距离。
  - single: 当前层使用 golden 输入时，板端当前层实际推理结果与 golden 结果对比的余弦距离和欧氏距离。

3.2.3 模型性能评估

接口`rknn.eval_perf()`会输出当前的硬件频率并获取模型的性能评估结果，`fix_freq`参数表示是否需要尝试对硬件（包括CPU/NPU/DDR）定频，如果要对硬件定频则设置成`True`，否则设置成`False`，默认值为`True`。若初始化时`rknn.init_runtime()`的`perf_debug`参数设置为`True`，将输出每一层的耗时情况和总耗时情况，若为`False`则只输出总耗时情况。

注意：

1.平台RV1103 / RV1103B / RV1106 / RV1106B不支持`perf_debug`为`True`模式，只能输出模型总耗时情况。

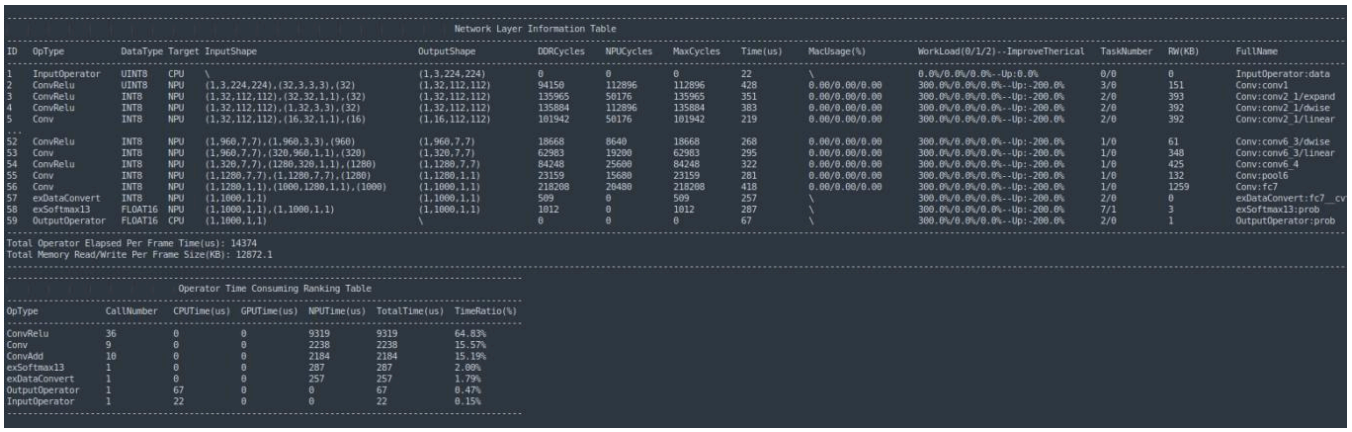
示例代码：

```
ret = rknn.init_runtime(target=platform, perf_debug=True)
perf_detail = rknn.eval_perf()
```

以RK3588为例，执行eval_perf后输出的硬件频率如下（不同固件支持的频率可能会有所差异），其中，由于RK3588 CPU是大小核架构，CPU频率栏会输出多个频率值，单位为KHz，而NPU和DDR频率单位都是Hz。

```
CPU Current Frequency List:
    - 1800000
    - 2256000
    - 2304000
NPU Current Frequency List:
    - 1000000000
DDR Current Frequency List:
    - 2112000000
```

性能评估结果如下：



Network Layer Information Table

ID	OpType	DataType	Target	InputShape	OutputShape	DDRCycles	NPUCycles	MaxCycles	Time(us)	MacUsage(%)	WorkLoad(0/1/2)--ImproveTherical	TaskNumber	RW(KB)	FullName
1	InputOperator	UINT8	CPU	\	(1,3,224,224)	0	0	0	22	\	0.0%/0.0%/0.0%--Up:0.0%	0/0	0	InputOperator:data
2	ConvRelu	UINT8	NPU	(1,3,224,224),(32,3,3,3),(32)	(1,32,112,112)	94150	112896	112896	428	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	3/0	151	Conv:conv1
3	ConvRelu	INT8	NPU	(1,32,112,112),(32,32,1,1),(32)	(1,32,112,112)	135965	50176	135965	351	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	2/0	393	Conv:conv2_1/expand
4	ConvRelu	INT8	NPU	(1,32,112,112),(1,32,3,3),(32)	(1,32,112,112)	135884	112896	135884	383	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	2/0	392	Conv:conv2_1/dwise
5	Conv	INT8	NPU	(1,32,112,112),(16,32,1,1),(16)	(1,16,112,112)	101942	50176	101942	219	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	2/0	392	Conv:conv2_1/linear
...														
52	ConvRelu	INT8	NPU	(1,960,7,7),(1,960,3,3),(960)	(1,960,7,7)	18668	8640	18668	268	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	1/0	61	Conv:conv6_3/dwise
53	Conv	INT8	NPU	(1,960,7,7),(320,960,1,1),(320)	(1,320,7,7)	62983	19200	62983	295	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	1/0	348	Conv:conv6_3/linear
54	ConvRelu	INT8	NPU	(1,320,7,7),(1280,320,1,1),(1280)	(1,1280,7,7)	84248	25600	84248	322	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	1/0	425	Conv:conv6_4
55	Conv	INT8	NPU	(1,1280,7,7),(1,1280,7,7),(1280)	(1,1280,1,1)	23159	15680	23159	281	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	1/0	132	Conv:pool6
56	Conv	INT8	NPU	(1,1280,1,1),(1000,1280,1,1),(1000)	(1,1000,1,1)	218208	20480	218208	418	0.00/0.00/0.00	300.0%/0.0%/0.0%--Up:-200.0%	1/0	1259	Conv:fc7
57	exDataConvert	INT8	NPU	(1,1000,1,1)	(1,1000,1,1)	509	0	509	257	\	300.0%/0.0%/0.0%--Up:-200.0%	2/0	0	exDataConvert:fc7__cv
58	exSoftmax13	FLOAT16	NPU	(1,1000,1,1),(1,1000,1,1)	(1,1000,1,1)	1012	0	1012	287	\	300.0%/0.0%/0.0%--Up:-200.0%	7/1	3	exSoftmax13:prob
59	OutputOperator	FLOAT16	CPU	(1,1000,1,1)	\	0	0	0	67	\	300.0%/0.0%/0.0%--Up:-200.0%	2/0	1	OutputOperator:prob

Total Operator Elapsed Per Frame Time(us): 14374
Total Memory Read/Write Per Frame Size(KB): 12872.1

Operator Time Consuming Ranking Table

OpType	CallNumber	CPUTime(us)	GPUTime(us)	NPUTime(us)	TotalTime(us)	TimeRatio(%)
ConvRelu	36	0	0	9319	9319	64.83%
Conv	9	0	0	2238	2238	15.57%
ConvAdd	10	0	0	2184	2184	15.19%
exSoftmax13	1	0	0	287	287	2.00%
exDataConvert	1	0	0	257	257	1.79%
OutputOperator	1	67	0	0	67	0.47%
InputOperator	1	22	0	0	22	0.15%

图3-3 性能评估结果

性能评估结果各字段说明如下表：

表3-2 性能评估结果各字段说明

字段	**详细说明**
ID	算子编号
OpType	算子类型
DataType	输入的数据类型
Target	算子运行的硬件（CPU/NPU/GPU）
InputShape	输入形状
OutputShape	输出形状
DDRCycles	DDR读写时钟周期数
NPUCycles	NPU计算时钟周期数
MaxCycles	DDR Cycles和NPU Cycles的最大值
Time(us)	算子计算耗时（us）
MacUsage(%)	MAC使用率
WorkLoad(0/1/2)	0/1/2核负载情况（仅RK3588平台）
WorkLoad(0/1)	0/1核负载情况（仅RK3576平台）

字段	详细说明
TaskNumber	NPU任务数
RW(KB)	读写的数据总量（KB）
FullName	算子全名
Total Operator Elapsed Per Frame Time(us)	模型推理的单帧总耗时
Total Memory Read/Write Per Frame Size(KB)	模型推理的单帧总带宽消耗
CallNumber	单帧内该算子运行次数
CPUTime(us)	单帧内该算子在CPU上的总耗时
GPUTime(us)	单帧内该算子在GPU上的总耗时
NPUTime(us)	单帧内该算子在NPU上的总耗时
TotalTime(us)	单帧内该算子的总耗时
TimeRatio(%)	单帧内该算子的总耗时与单帧总耗时的比值

3.2.4 模型内存评估

接口`rknn.eval_memory()`可以获取模型的内存消耗评估结果。初始化时`rknn.init_runtime()`的`eval_mem`参数设置为`True`，将输出各部分内存消耗情况。

示例代码：

```
ret = rknn.init_runtime(target=platform, eval_mem=True)
memory_detail = rknn.eval_memory()
```

内存评估结果如下：

```
==================================================
            Memory Profile Info Dump
==================================================
NPU model memory detail(bytes):
    Weight Memory: 8.67 MiB
    Internal Tensor Memory: 7.42 MiB
    Other Memory: 3.03 MiB
    Total Memory: 19.12 MiB

INFO: When evaluating memory usage, we need consider
the size of model, current model size is: 11.86 MiB
==================================================
```

内存评估结果各字段说明如下：

表3-3 内存评估结果各字段说明

字段	详细说明
Total Weight Memory	模型中权重的内存占用（MB）
Total Internal Tensor Memory	模型中间 tensor 内存占用（MB）
Other Memory	其他内存占用（例如寄存器配置、输入输出tensor）（MB）
Total Memory	模型的内存总占用（MB）

3.3 板端C API推理

本章节介绍通用C API接口的调用流程。零拷贝C API调用流程请参考章节5.2。

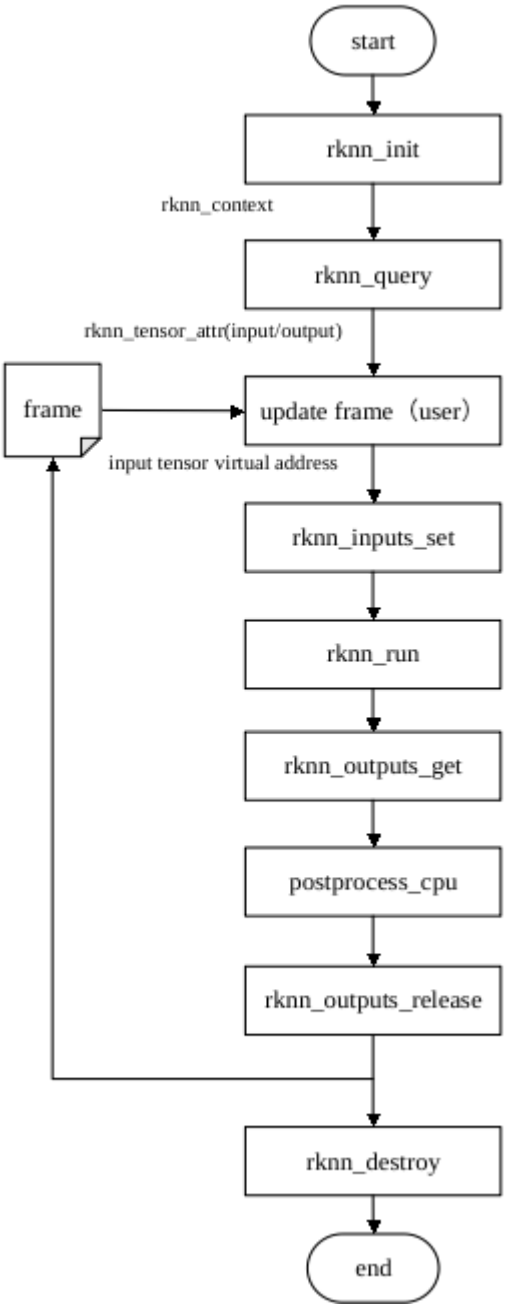


图3-4 通用API调用流程

RKNN通用API接口调用流程：

1. 调用`rknn_init()`接口初始化模型；

2. 调用`rknn_query()`接口查询模型的输入输出属性；
3. 对输入进行前处理；
4. 调用`rknn_inputs_set()`接口设置输入数据;
5. 调用`rknn_run()`接口进行模型推理；
6. 调用`rknn_outputs_get()`接口获取推理结果数据；
7. 对输出进行后处理；
8. 调用`rknn_outputs_release()`接口释放输出数据内存；
9. 调用`rknn_destroy()`接口销毁RKNN；

通用API调用流程如图3-4所示。

通用API调用流程示例代码：

```
// Init RKNN model
ret = rknn_init(&ctx, model, model_len, 0, NULL);

// Get Model Input Output Number
rknn_input_output_num io_num;
ret = rknn_query(ctx, RKNN_QUERY_IN_OUT_NUM, &io_num, sizeof(io_num));

// Get Model Input Info
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++)
{
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_ATTR, &(input_attrs[i]),
sizeof(rknn_tensor_attr));
}

// Get Model Output Info
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++)
{
    output_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_OUTPUT_ATTR, &(output_attrs[i]),
sizeof(rknn_tensor_attr));
}

rknn_input inputs[io_num.n_input];
rknn_output outputs[io_num.n_output];
memset(inputs, 0, sizeof(inputs));
memset(outputs, 0, sizeof(outputs));

// Pre-process
// Read Image
image_buffer_t src_image;
memset(&src_image, 0, sizeof(image_buffer_t));
ret = read_image(image_path, &src_image);
```

```
// Set Input Data
inputs[0].index = 0;
inputs[0].type = RKNN_TENSOR_UINT8;
inputs[0].fmt = RKNN_TENSOR_NHWC;
inputs[0].size = src_image.size;
inputs[0].buf = src_image.virt_addr;

ret = rknn_inputs_set(rknn_ctx, io_num.n_input, inputs);

// Run
ret = rknn_run(rknn_ctx, nullptr);

// Get Output Data
ret = rknn_outputs_get(rknn_ctx, io_num.n_output, outputs, NULL);

// Post-process
post_process(outputs, results);

// Release RKNN Output
rknn_outputs_release(rknn_ctx, io_num.n_output, outputs);

if (rknn_ctx != 0)
{
    rknn_destroy(rknn_ctx);
}
```

3.4 板端Python API推理

RKNN-Toolkit Lite2为用户提供板端模型推理的Python接口，方便用户使用Python语言进行AI应用开发。

注：在使用RKNN-Toolkit Lite2开发AI应用前，需要通过RKNN-Toolkit2将各深度学习框架导出的模型转成RKNN模型。模型转换详细方法请参考3.1章节。

3.4.1 系统依赖说明

使用RKNN-Toolkit Lite2需满足以下运行环境要求：

表3-4 RKNN-Toolkit Lite2运行环境

操作系统版本	Debian 10 / 11 (aarch64)
Python版本	3.7 / 3.8 / 3.9 / 3.10 / 3.11 / 3.12
Python依赖库	'numpy'、'ruamel.yaml'、'psutils'

3.4.2 工具安装

请通过`pip3 install`命令安装 RKNN-Toolkit Lite2。安装步骤如下：

- 如果系统中没有安装`python3/pip3`等程序，请先通过`apt-get`方式安装，参考命令如下：

```
sudo apt-get update
sudo apt-get install -y python3 python3-dev python3-pip gcc
```

注：安装部分依赖模块时，需要编译源码，此时将用到`python3-dev`和`gcc`，因此该步骤将这两个包也一起安装，避免后面安装依赖模块时编译失败。

- 安装依赖模块：`opencv-python`和`numpy`，参考命令如下：

```
sudo apt-get install -y python3-opencv
sudo apt-get install -y python3-numpy
```

注：

1. RKNN-Toolkit Lite2本身并不依赖`opencv-python`，但是在示例中需要使用该模块对图像进行处理。

2. 在Debian10固件上通过`pip3`安装`numpy`可能失败，建议用上述方法安装。

- 安装RKNN-Toolkit Lite2
  各平台的安装包都放在SDK的`rknn-toolkit-lite2/packages`文件夹下。进入packages文件夹，执行以下命令安装RKNN-Toolkit Lite2：

```
#  Python 3.7
pip3 install rknn_toolkit_lite2-x.y.z-cp37-cp37m-linux_aarch64.whl
#  Python 3.8
pip3 install rknn_toolkit_lite2-x.y.z-cp38-cp38-linux_aarch64.whl
# Python 3.9
pip3 install rknn_toolkit_lite2-x.y.z-cp39-cp39-linux_aarch64.whl
# Python 3.10
pip3 install rknn_toolkit_lite2-x.y.z-cp310-cp310-linux_aarch64.whl
# Python 3.11
pip3 install rknn_toolkit_lite2-x.y.z-cp311-cp311-linux_aarch64.whl
# Python 3.12
pip3 install rknn_toolkit_lite2-x.y.z-cp312-cp312-linux_aarch64.whl
```

3.4.3 基本使用流程

使用RKNN-Toolkit Lite2部署RKNN模型的基本流程如下图所示：

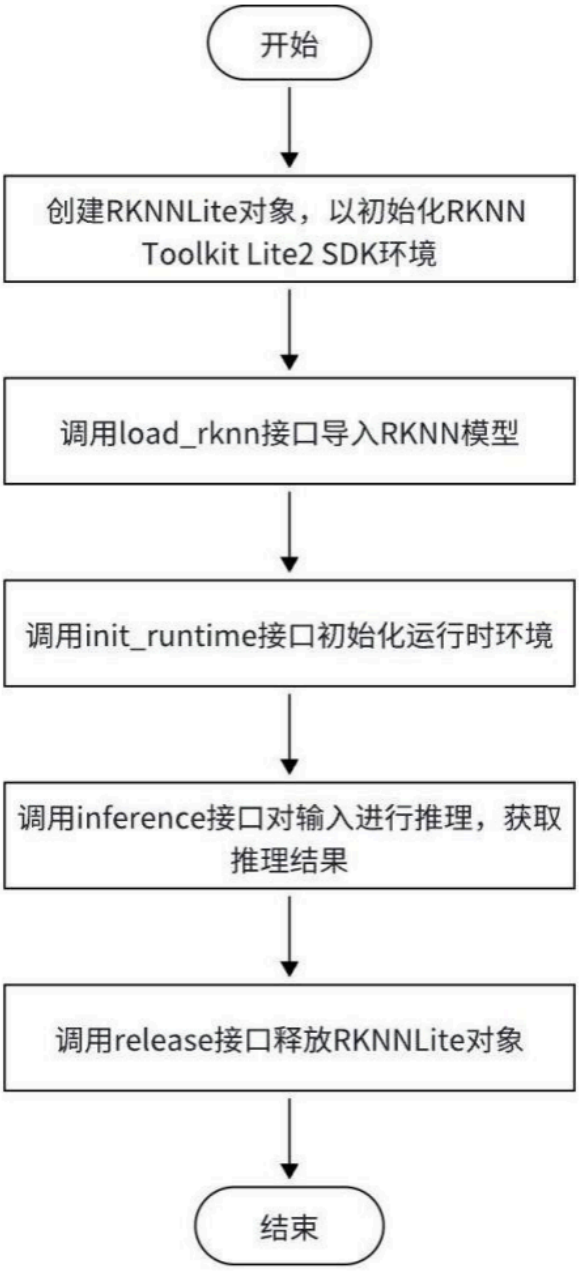


图3-5 RKNN-Toolkit Lite2基本使用流程

注：

1.在调用`inference`接口进行推理之前，需要获取输入数据，并做相应的预处理，然后根据输入信息设置`inference`接口中的各项参数。
2.在调用`inference`接口后，通常需要对推理结果进行相应的处理，以完成应用程序相关功能。

3.4.4 运行参考示例

在`SDK/rknn-toolkit-lite2/examples`目录，提供了一个基于RKNN-Toolkit Lite2开发的图像分类应用，该应用调用RKNN-Toolkit Lite2的接口加载Resnet18 RKNN模型，对输入图片进行预测，打印`top5`分类结果。

运行该示例的步骤：

1. 准备一块安装有RKNN-Toolkit Lite2的开发板；
2. 将`SDK/rknn-toolkit-lite2/examples`目录推到开发板上；
3. 在开发板上进入`examples/resnet18`目录，执行如下命令运行该示例：

```
python test.py
```

参考运行结果如下所示：

```
model: resnet18
-----TOP 5-----
[812] score:0.999680 class:"space shuttle"
[404] score:0.000249 class:"airliner"
[657] score:0.000013 class:"missile"
[466] score:0.000009 class:"bullet train, bullet"
[895] score:0.000008 class:"warplane, military plane"
```

3.4.5 RKNN-Toolkit Lite2 API详细说明

本章节将详细说明RKNN-Toolkit Lite2提供的所有API的用法。

3.4.5.1 RKNNLite初始化及对象释放

在使用RKNN-Toolkit Lite2时，需要先调用`RKNNLite()`方法初始化一个`RKNNLite`对象，并在用完后调用该对象的`rknn_lite.release()`方法将资源释放掉。

初始化`RKNNLite`对象时，可以设置`verbose`和`verbose_file`参数，以打印详细的日志信息。其中`verbose`参数指定是否要在屏幕上打印详细日志信息；如果设置了`verbose_file`参数，且`verbose`参数值为`True`，日志信息还将写到这个参数指定的文件中。

举例如下：

```
# 将详细的日志信息输出到屏幕，并写到inference.log文件中
rknn_lite = RKNNLite(verbose=True, verbose_file='./inference.log')
# 只在屏幕打印详细的日志信息
rknn_lite = RKNNLite(verbose=True)
…
rknn_lite.release()
```

3.4.5.2 加载RKNN模型

表3-5 load_rknn接口详细说明

API	load_rknn
描述	加载RKNN模型
参数	path: RKNN模型路径
返回值	0：加载成功；-1：加载失败。

举例如下：

```
# 从当前目录加载resnet_18.rknn模型
ret = rknn_lite.load_rknn('./resnet_18.rknn')
```

3.4.5.3 初始化运行时环境

在模型推理之前，必须先初始化运行时环境。

表3-6 init_runtime接口详细说明

API	init_runtime
描述	初始化运行时环境。
参数	core_mask：NPU工作核心配置模式。可选值如下： RKNNLite.NPU_CORE_AUTO: 自动调度模式，模型将以单核模式自动运行在当前空闲的NPU核上。 RKNNLite.NPU_CORE_0：模型运行在NPU Core0上。 RKNNLite.NPU_CORE_1：模型运行在NPU Core1上。 RKNNLite.NPU_CORE_2：模型运行在NPU Core2上。 RKNNLite.NPU_CORE_0_1：模型同时运行在NPU Core0和NPU Core1上。 RKNNLite.NPU_CORE_0_1_2：模型同时运行在NPU Core0，NPU Core1和NPU Core2上。 RKNNLite.NPU_CORE_ALL：模型同时运行在所有NPU核上。 默认值为`NPU_CORE_AUTO`，即默认使用的是自动调度模式。 注：该参数仅对RK3576 / RK3588有效。
返回值	0：初始化运行时环境成功；-1：初始化运行时环境失败。

举例如下：

```
# 初始化运行时环境
ret = rknn_lite.init_runtime(core_mask=RKNNLite.NPU_CORE_AUTO)
if ret != 0:
    print('Init runtime environment failed')
    exit(ret)
```

3.4.5.4 模型推理

表3-7 inference接口详细说明

API	inference
描述	对指定输入进行推理，返回推理结果。
参数	inputs：输入数据，如OpenCV读取的图片（如果输入是四维的，需要手动扩成4维）。类型是list，列表成员是ndarray。
	data_format：数据排列方式，类型是list，对于每个输入可选值"nhwc", "nchw"，只对四维输入有效。默认值为None。如果不填写该参数，对于4维输入，数据buffer应按照NHWC排列，对于非4维输入，数据buffer应按照模型输入要求的格式排列。如果要填写该参数，对于非4维输入，数据buffer应按照模型输入要求的格式排列（不管填"nhwc"还是"nchw"）；对于4维输入，数据buffer应按照该参数设置的格式排列；对于多输入模型，填写该参数时要包括所有输入。

API	inference
返回值	results：推理结果，类型是list，列表成员是ndarray。

以分类模型为例，模型推理代码参考如下：

```
# Get input data
img = cv2.imread('./space_shuttle_224.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = np.expand_dims(img, 0)
# Inference
outputs = rknn_lite.inference(inputs=[img])
# Show the classification results
show_top5(outputs)
```

3.4.5.5 查询SDK版本

表3-8 get_sdk_version接口详细说明

API	get_sdk_version
描述	获取Runtime， 驱动和RKNN模型版本信息。注：使用该接口前必须完成模型加载和初始化运行环境。
参数	无
返回值	sdk_version：runtime， 驱动版本信息。类型为字符串。

举例如下：

```
# 获取SDK版本信息
sdk_version = rknn_lite.get_sdk_version()
```

返回的SDK信息参考如下：

```
=============================================
RKNN VERSION:
  API: 1.5.2 (71720f3fc@2023-08-21T09:29:52)
  DRV: 0.7.2
=============================================
```

3.4.5.6 查询模型可运行平台

表3-9 list_support_target_platform接口详细说明

API	list_support_target_platform
描述	查询给定RKNN模型可运行的芯片平台。

API	list_support_target_platform
参数	rknn_model：RKNN模型路径。如果不指定模型路径，则按类别打印RKNN-Toolkit Lite2当前支持的芯片平台。
返回值	support_target_platform: 返回模型可运行的芯片平台。如果RKNN模型路径为空或不存在，返回None.

参考代码如下：

```
rknn_lite.list_support_target_platform(rknn_model='mobilenet_v1.rknn')
```

参考结果如下：

```
**********************************************
Target platforms filled in RKNN model:         ['rk3566']
Target platforms supported by this RKNN model: ['RK3566', 'RK3568']
**********************************************
```

3.5 矩阵乘法接口

Matmul API是Runtime提供的一套单独的C API，用于在NPU上运行矩阵乘法运算，RK2118 / RV1103系列 / RV1106系列暂不支持。矩阵乘法是线性代数中的一种重要操作，该操作定义如下：**C=AB**。其中，A是一个M×K矩阵，B是一个K×N矩阵，C是一个M×N矩阵。

3.5.1 主要用途和特点

Matmul API多用于深度学习中的参数计算任务，例如，在当前被广泛应用的Transformer模型结构的主要模块（自注意力机制和前馈神经网络层）中大量使用了矩阵乘法进行计算，因此矩阵乘法的运算效率对Transfomer模型的整体性能至关重要。Matmul API具有以下特点：

- **高效**：底层使用RKNPU实现，具有高性能低功耗的优点。
- **灵活**：无需加载RKNN模型，支持int4、int8和float16三种边缘端计算常用的输入数据类型，提供单独的内存分配接口或使用外部内存的机制，用户可管理和复用矩阵的输入输出内存。

3.5.2 Matmul API使用流程

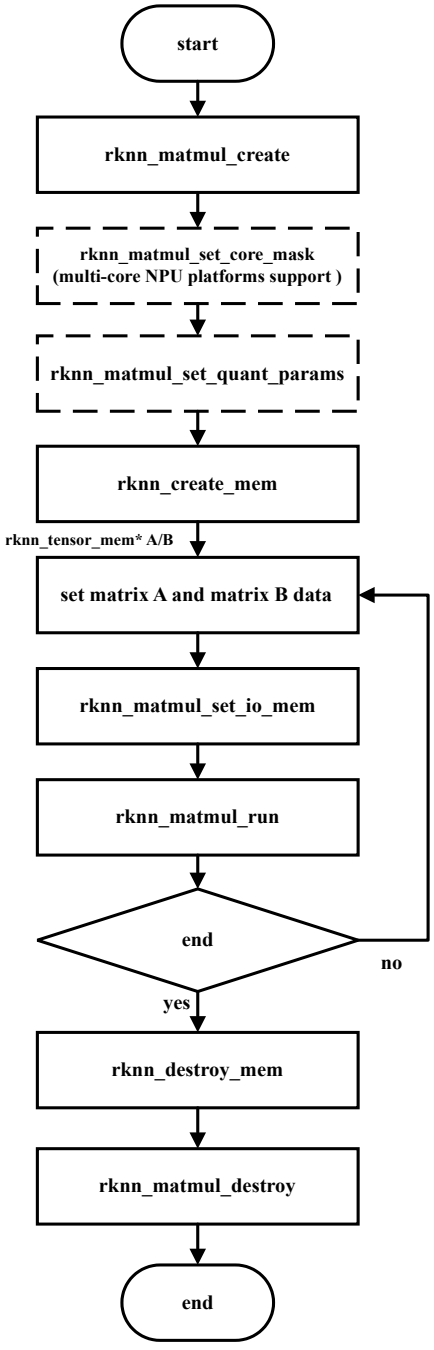


图3-6 Matmul API基础调用流程

首先，Matmul API的结构体和接口位于rknn_matmul_api.h头文件，开发者程序编译时需要包含该头文件。Matmul API基础用法的流程如上图所示。

使用Matmul API通常包括以下步骤：

1. 创建上下文：设置rknn_matmul_info结构体，包含M、K、N、输入和输出矩阵的数据类型、输入和输出矩阵使用的数据排列方式等信息，然后，调用rknn_matmul_create接口初始化上下文。在初始化后，获取以rknn_matmul_io_attr结构体指针，它包含了输入和输出矩阵tensor信息。
2. 指定运行NPU核（仅多核NPU的芯片平台有效）：调用rknn_matmul_set_core_mask，设置掩码来指定某一个NPU核做运算，多核NPU的芯片平台不调用该接口的默认行为是自动选择空闲的核心。

3. 设置矩阵A、B和C的量化参数：调用rknn_matmul_set_quant_params传入包含对应矩阵量化参数值的rknn_quant_params结构体，不调用该接口的默认行为是各个矩阵的scale=1.0, zero_point=0。
4. 创建输入和输出内存：调用rknn_create_mem接口，根据输入和输出矩阵Tensor信息中的大小创建内存。
5. 填充输入数据：根据形状和数据类型填充输入矩阵A和B的数据，并且以量化后的值设置量化参数。
6. 设置输入和输出内存：调用rknn_matmul_set_io_mem将填充好数据的输入矩阵记录到上下文中，输出内存也同样记录到上下文中。除了记录内存地址外，该接口还会按照所设置的layout对数据做重新排列，必须在填充或更新输入数据后调用，与零拷贝API中的rknn_set_io_mem接口行为有区别。
7. 执行矩阵乘法运算：设置好输入和输出内存后，调用rknn_matmul_run执行矩阵乘法运算。
8. 处理输出：执行矩阵乘法运算后，从输出内存中按照所设置的layout来读取结果。
9. 销毁资源：执行结束后，调用rknn_destroy_mem和rknn_matmul_destroy分别销毁内存和上下文资源。

3.5.3 矩阵乘法高级用法

在创建上下文时，要求用户设置`rknn_matmul_info`结构体，`rknn_matmul_info`表示用于执行矩阵乘法的规格信息，它包含了矩阵乘法的规模、输入和输出矩阵的数据类型和数据排列。其中，B_layout和AC_layout用于设置高性能的数据排列方式。具体的结构体定义如下表所示：

表3-10 rknn_matmul_info结构体定义

成员变量	数据类型	含义
M	int32_t	A矩阵的行数
K	int32_t	A矩阵的列数
N	int32_t	B矩阵的列数
type	rknn_matmul_type	输入输出矩阵的数据类型： RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT32：表示矩阵A和B是float16类型，矩阵C是float32类型； RKNN_INT8_MM_INT8_TO_INT32：表示矩阵A和B是int8类型，矩阵C是int32类型； RKNN_INT8_MM_INT8_TO_INT8：表示矩阵A、B和C是int8类型； RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT16：表示矩阵A、B和C是float16类型； RKNN_FLOAT16_MM_INT8_TO_FLOAT32：表示矩阵A是float16类型，矩阵B是int8类型，矩阵C是float32类型； RKNN_FLOAT16_MM_INT8_TO_FLOAT16：表示矩阵A是float16类型，矩阵B是int8类型，矩阵C是float16类型； RKNN_FLOAT16_MM_INT4_TO_FLOAT32：表示矩阵A是float16类型，矩阵B是int4类型，矩阵C是float32类型； RKNN_FLOAT16_MM_INT4_TO_FLOAT16：表示矩阵A是float16类型，矩阵B是int4类型，矩阵C是float16类型； RKNN_INT8_MM_INT8_TO_FLOAT32：表示矩阵A和B是int8类型，矩阵C是float32类型; RKNN_INT4_MM_INT4_TO_INT16：表示矩阵A和B是int4类型，矩阵C是int16类型； RKNN_INT8_MM_INT4_TO_INT32：表示矩阵A是int8类型，B是int4类型，矩阵C是int32类型

成员变量	数据类型	含义
B_layout	int16_t	矩阵B的数据排列方式。 RKNN_MM_LAYOUT_NORM：表示矩阵B按照原始形状排列，即KxN的形状排列； RKNN_MM_LAYOUT_NATIVE：表示矩阵B按照高性能形状排列； RKNN_MM_LAYOUT_TP_NORM：表示矩阵B按照Transpose后的形状排列，即NxK的形状排列
B_quant_type	int16_t	矩阵B的量化方式类型。 RKNN_QUANT_TYPE_PER_LAYER_SYM：表示矩阵B按照Per-Layer方式对称量化； RKNN_QUANT_TYPE_PER_LAYER_ASYM：表示矩阵B按照Per-Layer方式非对称量化； RKNN_QUANT_TYPE_PER_CHANNEL_SYM：表示矩阵B按照Per-Channel方式对称量化； RKNN_QUANT_TYPE_PER_CHANNEL_ASYM：表示矩阵B按照Per-Channel方式非对称量化； RKNN_QUANT_TYPE_PER_GROUP_SYM：表示矩阵B按照Per-Group方式对称量化； RKNN_QUANT_TYPE_PER_GROUP_ASYM：表示矩阵B按照Per-Group方式非对称量化
AC_layout	int16_t	矩阵A和C的数据排列方式。 RKNN_MM_LAYOUT_NORM：表示矩阵A和C按照原始形状排列； RKNN_MM_LAYOUT_NATIVE：表示矩阵A和C按照高性能形状排列； 详细的排列说明可以参考3.5.4
AC_quant_type	int16_t	矩阵A和C的量化方式类型。 RKNN_QUANT_TYPE_PER_LAYER_SYM：表示矩阵A和C按照Per-Layer方式对称量化； RKNN_QUANT_TYPE_PER_LAYER_ASYM：表示矩阵A和C按照Per-Layer方式非对称量化
iommu_domain_id	int32_t	矩阵上下文所在的IOMMU地址空间域的索引。IOMMU地址空间与上下文一一对应，每个IOMMU地址空间大小为4GB。该参数主要用于矩阵A、B和C的参数规格较大，某个域内NPU分配的内存超过4GB以后需切换另一个域时使用。
group_size	int16_t	一个组的元素数量，仅分组量化开启后生效。
reserved	int8_t[]	预留字段

其中，矩阵A的原始形状是MxK，矩阵B的原始形状是KxN，矩阵C的原始形状是MxN。

3.5.3.1 指定量化参数的矩阵乘法

如果矩阵乘积的高比特数据需要根据量化参数量化成低比特数据，例如，矩阵A和B是int8类型，矩阵C是int8类型，需要设置矩阵的量化参数，量化参数由scale和zero_point组成，用rknn_quant_params结构体表示，通过rknn_matmul_set_quant_params接口设置矩阵A和C的量化参数，量化参数设置成功后，Per-Channel量化方式下的矩阵B的量化参数可以通过rknn_matmul_get_quant_params接口读取。rknn_quant_params数据结构定义如下表所示：

表3-11 rknn_quant_params结构体定义

成员变量	数据类型	含义
name	char[]	矩阵的名称
scale	float*	量化参数scale数组的首地址
scale_len	int32_t	量化参数scale数组的长度
zp	int32_t*	量化参数zero_point数组的首地址
zp_len	int32_t	量化参数zero_point数组的长度

Matmul的量化方式包含Per-Layer方式、Per-Channel方式以及Per-Group方式三种，量化参数差别如下：

- Per-Layer方式: scale和zp数组的长度为1。
- Per-Channel方式: scale和zp数组的长度为N。
- Per-Group方式: scale和zp数组的长度为N*K/group_size，每个组的元素数量由group_size指定，同一个组内的元素共享相同的量化参数，量化参数数组存储方式上，先存储K/group_size个组的量化参数，再存储N个量化参数。

每种量化方式存在对称量化和非对称量化两种方式，对称量化表示zp全为0，非对称量化表示zp不全为0。

3.5.3.2 动态shape输入的矩阵乘法

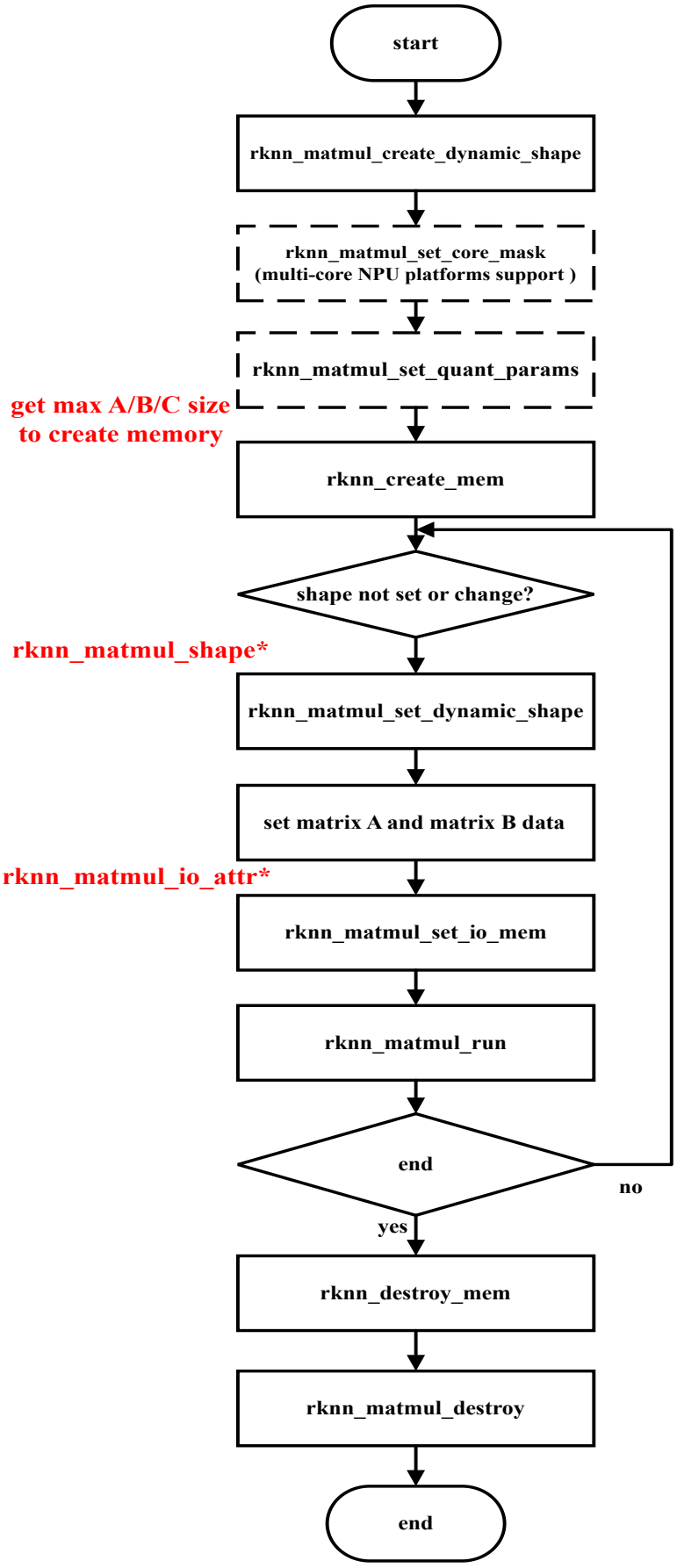


图3-7 动态shape输入的Matmul API调用流程

动态shape输入的Matmul API用法的流程如上图所示。使用动态shape Matmul API与基础的Matmul API的流程差异如下：

1. 创建上下文：配置shape数量和所需的shape，设置rknn_matmul_info结构体，包含输入和输出矩阵的数据类型、输入和输出矩阵使用的数据排列方式等信息。**注意，动态shape Matmul接口参数rknn_matmul_info中的M、K、N不需要设置**。然后，调用rknn_matmul_create_dynamic_shape接口初始化上下文。在初始化后，获取rknn_matmul_io_attr结构体数组，它包含了所有shape配置下的输入和输出矩阵Tensor信息。
2. 创建输入和输出内存：首先从rknn_matmul_io_attr结构体数组获取最大的Tensor大小，调用rknn_create_mem接口，使用最大的输入和输出矩阵Tensor大小创建内存。
3. 设置输入shape: 通过调用rknn_matmul_set_dynamic_shape接口，传入 rknn_matmul_shape指针，设置当前推理过程使用的shape。
4. 设置输入和输出内存：调用rknn_matmul_set_io_mem将填充好数据的输入矩阵记录到上下文中，输出内存也同样记录到上下文中。传入的rknn_tensor_mem*参数是前面创建的最大size的输入和输出内存，rknn_matmul_tensor_attr*参数是当前shape对应的矩阵属性信息。

3.5.4 高性能的数据排列方式

由于NPU是专用的硬件架构，读取MxK和KxN这种原始形状的数据不是最高效的，同样的，写入MxN形状的C矩阵也不是最高效的，用户使用特殊的数据排列方式可以实现更高的性能。**AC_layout参数控制矩阵A和C是否使用高性能数据排列，B_layout参数控制矩阵B是否使用高性能数据排列**。
假设矩阵A的原始形状是MxK，矩阵B的原始形状是KxN，矩阵C的原始形状是MxN，要求的数据排列方式如下：

1. 若AC_layout=RKNN_MM_LAYOUT_NORM且B_layout=RKNN_MM_LAYOUT_NORM，矩阵A的形状为[M,K]，矩阵B的形状为[K,N]，矩阵C的形状为[M,N]。
2. 若AC_layout=RKNN_MM_LAYOUT_NATIVE且B_layout=RKNN_MM_LAYOUT_NATIVE，不同芯片平台和数据类型下矩阵A、B和C的高性能数据排列如下表所示（表中除法结果都是上取整，多出部分用0补齐）：
3. B_layout=RKNN_MM_LAYOUT_NORM和B_layout=RKNN_MM_LAYOUT_TP_NORM的区别在于，前者矩阵B的形状为[K，N]，后者矩阵B的形状为[N，K]，后者的执行效率更优。

表3-12 各个芯片平台矩阵A、B和C的高性能形状

	RK3562	RK3566/RK3568	RK3576/RV1126B	RK3588
A形状(int4)	暂不支持	暂不支持	[K/32,M,32]	[K/32,M,32]
B形状(int4)	暂不支持	暂不支持	[N/64,K/32,64,32]*	[N/64,K/32,64,32]
C形状(int16)	暂不支持	暂不支持	[N/8,M,8]	[N/8,M,8]
A形状(int8)	[K/16,M,16]	[K/8,M,8]	[K/16,M,16]	[K/16,M,16]
B形状(int8)	[N/16,K/32,16,32]	[N/16,K/32,16,32]	[N/32,K/32,32,32]*	[N/32,K/32,32,32]
C形状(int32)	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]
A形状(float16)	[K/8,M,8]	[K/4,M,4]	[K/8,M,8]	[K/8,M,8]
B形状(float16)	[N/8,K/32,8,32]	[N/8,K/16,8,16]	[N/16,K/32,16,32]	[N/16,K/32,16,32]
C形状(float32)	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]

Note

> 上表中的数据排列方式是A和B“**同型**”(即A和B相同数据类型)情况下的高性能形状。RK3576/RV1126B支持“**异型**”输入，即允许A和B有不同的数据类型，在“异型”输入情况下，B的高性能数据排列与C类型对应的“同型”情况下B的形状相同。例如，A是int4类型，B是float16类型，C是float32类型，则B的高性能数据数据排列是[N/16,K/32,16,32]。`rknn_B_normal_layout_to_native_layout`接口提供了对B的原始形状转换成高性能排列的功能。

以RK3566/RK3568平台，int8数据类型为例，矩阵A从[M,K]变换到[K/8,M,8]的元素对应关系如下图所示：

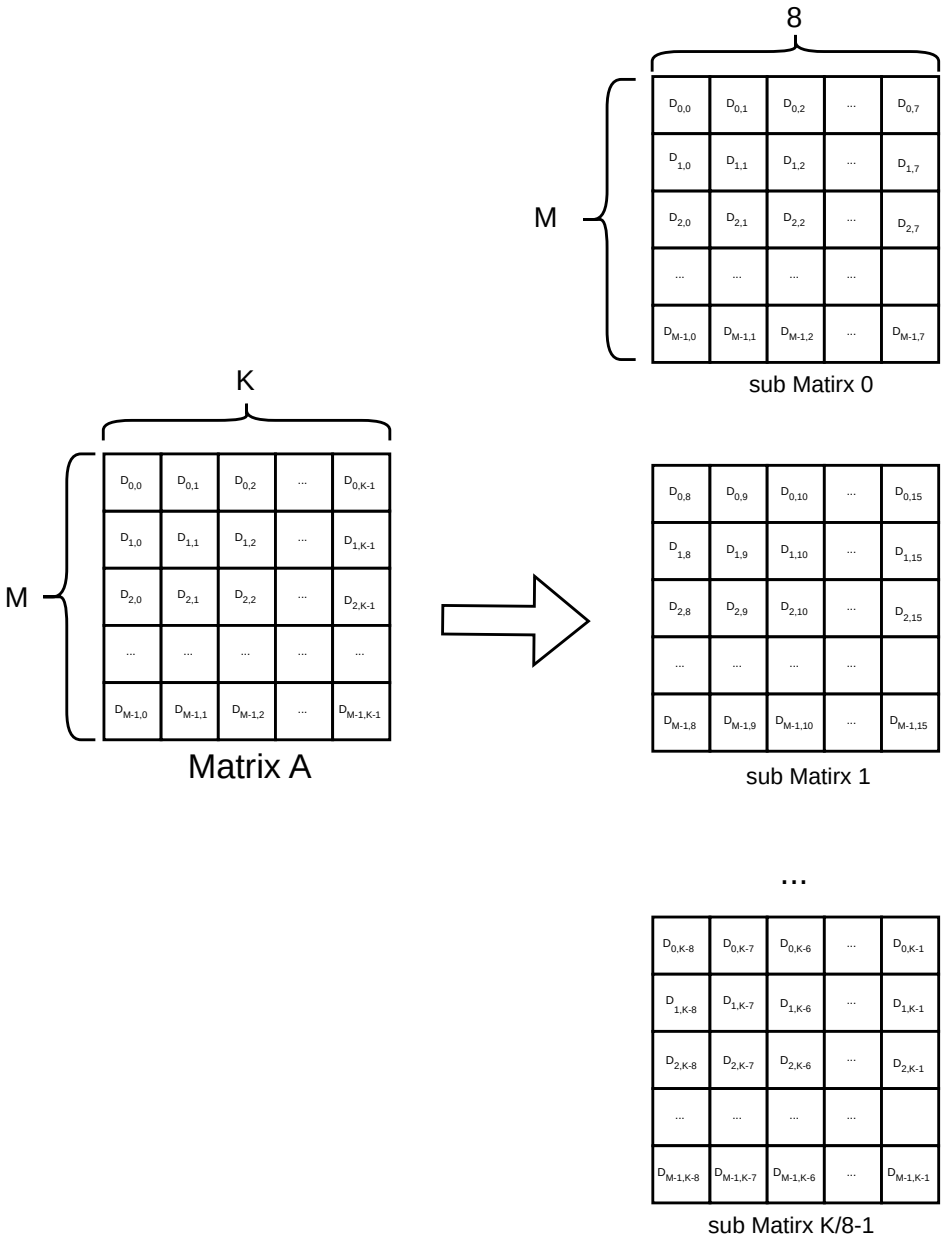


图3-8 int8类型矩阵A从[M,K]变换到[K/8,M,8]的元素对应关系图

其中，矩阵A第(i，j)元素对应的数据为Di,j，左图是shape=[M,K]的矩阵，右图每个小矩阵shape=[M,8]，从上到下一共K/8个。

矩阵A或C从原始形状转换成高性能形状的示例代码如下：

```
template <typename Ti, typename To>
void norm_layout_to_perf_layout(Ti *src, To *dst, int32_t M, int32_t K, int32_t subK){
    int outter_size = (int)std::ceil(K * 1.0f / subK);
```

```
    for (int i = 0; i < outter_size; i++)
    {
        for (int m = 0; m < M; m++)
        {
            for (int j = 0; j < subK; j++)
            {
                int ki = i * subK + j;
                if (ki >= K)
                {
                  dst[i * M * subK + m * subK + j] = 0;
                }
                else
                {
                  dst[i * M * subK + m * subK + j] = src[m * K + ki];
                }
            }
        }
    }
}
```

以RK3566/RK3568平台，int8数据类型的矩阵B从[K,N]变换到[N/16,K/32,16,32]的元素对应关系如下图所示：

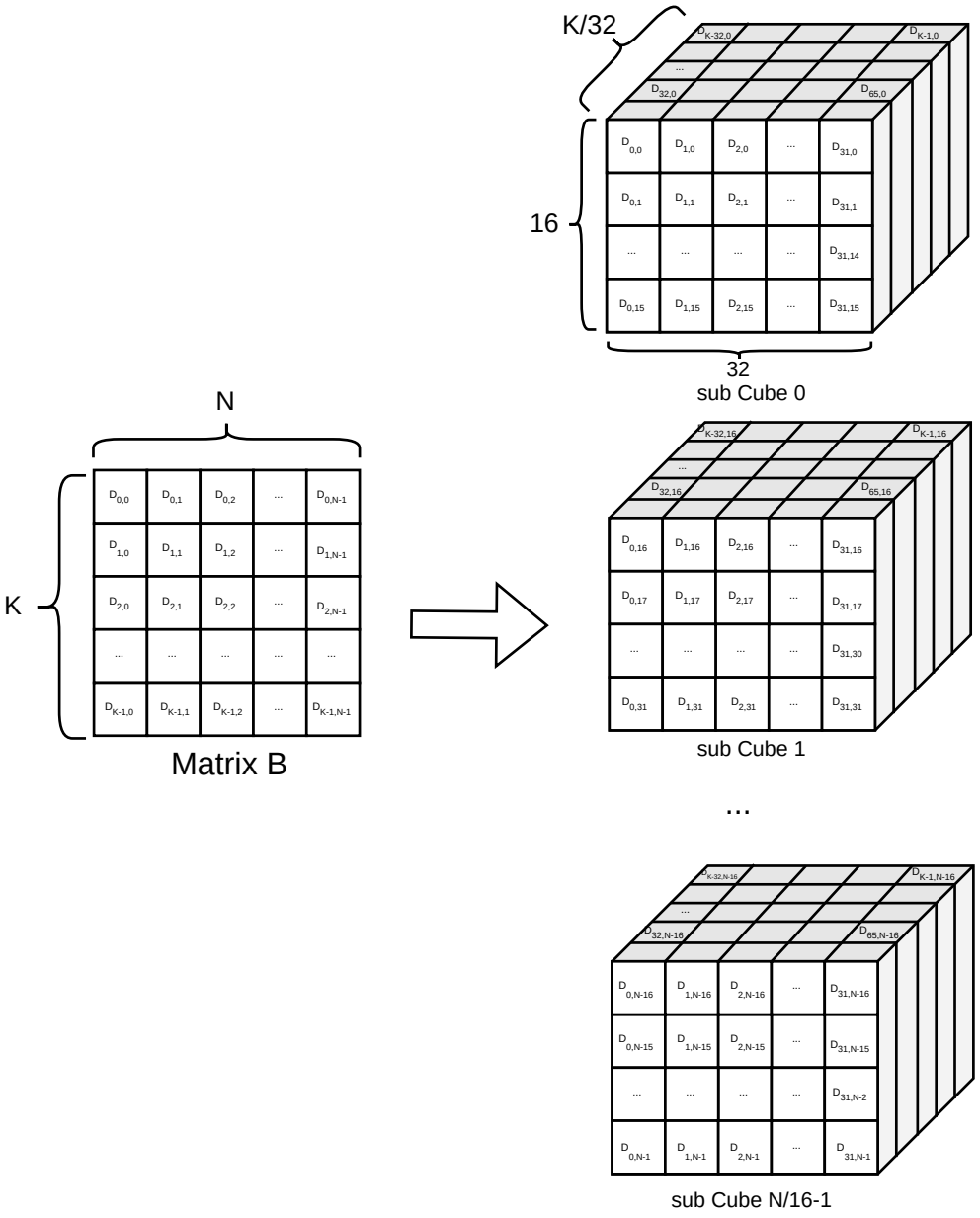


图3-9 int8类型矩阵B从[K,N]变换到[N/16,K/32,16,32]的元素对应关系图

其中，矩阵B第(i，j)元素对应的数据为Di,j，左图是shape=[K,N]的矩阵，右图每个小立方体shape=[K/32,16,32]，从上到下一共N/16个。

矩阵B从原始形状转换成高性能形状的示例代码如下：

```
template <typename Ti, typename To>
voi norm_layout_to_native_layout(Ti *src, To *dst, int32_t K, int32_t N, int32_t subN,
int32_t subK)
{
    int N_remain = (int)std::ceil(N * 1.0f / subN);
    int K_remain = (int)std::ceil(K * 1.0f / subK);
    for (int i = 0; i < N_remain; i++)
    {
        for (int j = 0; j < K_remain; j++)
        {
            for (int n = 0; n < subN; n++)
```

```
            {
                int ni = i * subN + n;
                for (int k = 0; k < subK; k++)
                {
                    int ki = j * subK + k;
                    if (ki < K && ni < N)
                    {
                        dst[((i * K_remain + j) * subN + n) * subK + k] = src[ki * N + ni];
                    }
                    else
                    {
                        dst[((i * K_remain + j) * subN + n) * subK + k] = 0;
                    }
                }
            }
        }
    }
}
```

> **Note**
>
> Matmul API完整示例代码见 https://github.com/airockchip/rknn-toolkit2/tree/master/rknpu2/examples/rknn_matmul_api_demo。

3.5.4.1 矩阵规格限制

Matmul API是基于NPU的硬件架构实现，受硬件规格限制。在参数上有如下几点限制：

- AC_layout和B_layout可独立设置。所有硬件平台上的AC_layout和B_layout都支持RKNN_MM_LAYOUT_NORM或者RKNN_MM_LAYOUT_TP_NORM，除了RK3566/RK3568平台外，其他平台都支持RKNN_MM_LAYOUT_TP_NORM。
- Matmul接口支持多种输入和输出的数据位宽，目前各个平台支持的矩阵数据类型组合如下表：

表3-13 Matmul接口支持的矩阵A、B和C的数据类型组合

序号	RK356X	RK3576/RV1126B	RK3588
1	RKNN_INT8_MM_INT8_TO_INT32	RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT16	RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT16
2	RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT32	RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT32	RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT32
3		RKNN_INT8_MM_INT8_TO_INT32	RKNN_INT8_MM_INT8_TO_INT32
4		RKNN_FLOAT16_MM_INT8_TO_FLOAT32	RKNN_INT8_MM_INT8_TO_FLOAT32
5		RKNN_FLOAT16_MM_INT4_TO_FLOAT16	RKNN_INT8_MM_INT8_TO_INT8
6		RKNN_FLOAT16_MM_INT4_TO_FLOAT32	RKNN_INT4_MM_INT4_TO_INT16
7		RKNN_INT8_MM_INT4_TO_INT32	

其中，RK356X表示RK3562/RK3566/RK3568等平台，float16浮点格式遵循IEEE-754标准，具体格式请参考 [IEEE-754 half] (https://en.wikipedia.org/wiki/Half-precision_floating-point_format)。

- Matmul中K和N大小限制如下表：

表3-14 各个芯片平台K和N的大小限制

	RK3562	RK3566/RK3568	RK3576/RV1126B	RK3588
K大小限制(int4)	暂不支持	暂不支持	*	32对齐
K大小限制(int8)	K <= 10240且K mod 32 = 0	K <= 10240且K mod 32 = 0	32对齐	32对齐
K大小限制(float16)	K <= 10240且K mod 32 = 0	K <= 10240且K mod 32 = 0	32对齐	32对齐
N大小限制(int4)	暂不支持	暂不支持	*	N <= 8192且N mod 64 = 0
N大小限制(int8)	N <= 4096且N mod 16 = 0	N <= 4096且N mod 16 = 0	N <= 4096且N mod 32 = 0	N <= 4096且N mod 32 = 0
N大小限制(float16)	N <= 4096且N mod 16 = 0	N <= 4096且N mod 16 = 0	N <= 4096且N mod 32 = 0	N <= 4096且N mod 32 = 0

- > **Note**
  >
  > *表示大小和对齐限制由同平台下A的数据类型对应的B限制相同。例如，RKNN_FLOAT16_MM_INT4_TO_FLOAT32组合类型B的大小和对齐限制与RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT32组合相同。

对于RK3588：

当K > 8192, B 会被分成T段，

```
int T = std::ceil(K / 8192);
```

例如: normal layout -> native layout，
K = 20488, N = 4096, T = 3, 数据会被分成3段，

```
subN = rknn_matmul_io_attr.B.dims[2];
subK = rknn_matmul_io_attr.B.dims[3];
```

K,N转变过程如下：

```
                                        (8196, 4096)   (4096 / subN, 8196 / subK, subN,
subK)
 (K, N) = (20488, 4096)->(8196, 4096)->(4096 / subN, 8196 / subK, subN, subK)
                 normal layout  (4096, 4096)    (4096 / subN, 4096 / subK, subN, subK)
                                       T normal layout                  T native layout
```

推荐使用rknn_B_normal_layout_to_native_layout接口进行直接数据转换。

对于RK3576/RV1126B：
当K > 4096, B 会被分成T段，

```
int T = std::ceil(K / 4096);
```

例如: normal layout -> native layout，
K = 10240, N = 2048, T = 3, 数据会被分成3段，

```
subN = rknn_matmul_io_attr.B.dims[2];
subK = rknn_matmul_io_attr.B.dims[3];
```

K,N转变过程如下：

```
                                        (4096, 2048)   (2048 / subN, 4096 / subK, subN,
subK)
(K, N) = (10240, 2048) ->(4096, 2048)->(2048 / subN, 4096 / subK, subN, subK)
                 normal layout   (2048, 2048)    (2048 / subN, 2048 / subK, subN, subK)
                                       T normal layout                    T native layout
```

推荐使用rknn_B_normal_layout_to_native_layout接口进行直接数据转换。

4 示例

RKNN提供了不同模型的参考示例，包括MobileNet图像分类、YOLOv5目标检测等，代码工程位于https://github.com/airockchip/rknn_model_zoo/tree/main/examples目录下。

本章节以如下环境为例介绍如何运行参考示例：

- PC端：操作系统为Ubuntu 22.04，Python环境为Miniforge创建的Python3.8环境；
- 开发板为RK3588 Linux平台为例。

可以参考第二章准备开发环境，其他平台的部署流程可参考`<Rockchip_RKNPU_Quick_Start_RKNN_SDK>`文档。

4.1 MobileNet模型部署示例

本章节以MobileNet模型部署为例，介绍如何快速上手模型转换、模型连板运行、模型评估和模型板端部署。

4.1.1 模型转换

1.进入`rknn_model_zoo/examples/mobilenet/python`目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型转换并进行图片推理

```
python mobilenet.py --model ../model/mobilenetv2-12.onnx --target rk3588
```

执行该命令后模型是在电脑模拟器上进行推理，转换后的模型默认保存路径为`rknn_model_zoo/examples/mobilenet/model/mobilenet_v2.rknn`。

4.1.2 模型连板运行

1.进入`rknn_model_zoo/examples/mobilenet/python`目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型连板运行

```
python mobilenet.py --target rk3588 --npu_device_test
```

执行该命令后模型通过连板的方式在板端上进行推理。输出结果如下：

```
-----TOP 5-----
[494] score=0.99 class="n03017168 chime, bell, gong"
[469] score=0.00 class="n02939185 caldron, cauldron"
[653] score=0.00 class="n03764736 milk can"
[747] score=0.00 class="n04023962 punching bag, punch bag, punching ball, punchball"
[505] score=0.00 class="n03063689 coffeepot"
```

4.1.3 模型评估

RKNN提供(模拟器和板端)精度评估、耗时评估和内存评估的功能，辅助RKNN模型的优化和部署。

4.1.3.1 精度评估

1.进入`rknn_model_zoo/examples/mobilenet/python`目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型连板精度分析

```
python mobilenet.py --target rk3588 --accuracy_analysis --npu_device_test
```

模型连板精度分析的输出结果如下：

```
# simulator_error: calculate the output error of each layer of the simulator (compared to
the 'golden' value).
#            entire: output error of each layer between 'golden' and 'simulator', these
errors will accumulate layer by layer.
#            single: single-layer output error between 'golden' and 'simulator', can better
reflect the single-layer accuracy of the simulator.
layer_name                              simulator_error
                                  entire                 single
                                cos      euc           cos      euc
----------------------------------------------------------------
...
[Conv] 464                  0.99202 | 4.1079    0.99998 | 0.1981
[Conv] output_conv          0.99308 | 13.235    0.99992 | 1.4133
[Reshape] output_int8       0.99308 | 13.235    0.99993 | 1.3043
[exDataConvert] output      0.99308 | 13.235    0.99993 | 1.3043


# runtime_error: calculate the output error of each layer of the runtime.
#          entire: output error of each layer between 'golden' and 'runtime', these errors
will accumulate layer by layer.
#     single_sim: single-layer output error between 'simulator' and 'runtime', can better
reflect the single-layer accuracy of runtime.
layer_name                              runtime_error
                                  entire               single_sim
                                cos      euc           cos      euc
----------------------------------------------------------------
...
[Conv] 464                    0.99210 | 4.2718    1.00000 | 0.0
[Conv] output_conv            0.99203 | 14.847    1.00000 | 0.2007
[Reshape] output_int8
[exDataConvert] output        0.99203 | 14.847    1.00000 | 0.0
```

4.1.3.2 耗时评估

1.进入`rknn_model_zoo/examples/mobilenet/python`目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型耗时评估

```
python mobilenet.py --target rk3588 --eval_perf
```

模型耗时评估的输出结果如下：

```
-------------------------------------------------------------
         Network Layer Information Table
-------------------------------------------------------------
ID   OpType           DataType  Target   Time(us)   ...
-------------------------------------------------------------
1    InputOperator    UINT8     CPU      17
2    ConvClip         UINT8     NPU      331
3    ConvClip         INT8      NPU      429
4    Conv             INT8      NPU      292
....
55   Conv             INT8      NPU      374
56   Reshape          INT8      CPU      61
57   OutputOperator   INT8      CPU      11
-------------------------------------------------------------
Total Operator Elapsed Per Frame Time(us): 12631
Total Memory Read/Write Per Frame Size(KB): 10563
-------------------------------------------------------------

-------------------------------------------------------------
          Operator Time Consuming Ranking Table
-------------------------------------------------------------
OpType      CallNumber ... NPUTime(us) TotalTime(us) TimeRatio(%)
-------------------------------------------------------------
ConvClip         35            8436         8436           66.79%
Conv             9             2093         2093           16.57%
ConvAdd          10            2013         2013           15.94%
Reshape          1             0            61              0.48%
InputOperator    1             0            17              0.13%
OutputOperator   1             0            11              0.09%
-------------------------------------------------------------
```

4.1.3.3 内存评估

1.进入`rknn_model_zoo/examples/mobilenet/python`目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型内存评估

```
python mobilenet.py --target rk3588 --eval_memory
```

模型内存评估的输出结果如下：

```
==================================
            Memory Profile Info Dump
==================================
NPU model memory detail(bytes):
    Weight Memory: 3.53 MiB
    Internal Tensor Memory: 1.53 MiB
    Other Memory: 377.19 KiB
    Total Memory: 5.43 MiB

INFO: When evaluating memory usage, we need consider
the size of model, current model size is: 3.98 MiB
==================================
```

4.1.4 板端部署

1.在rknn_model_zoo工程下的build-linsx.sh 脚本中指定gcc交叉编译器路径

```
GCC_COMPILER=~/opts/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu
```

有关gcc交叉编译器的下载和安装方法可参考<`Rockchip_RKNPU_Quick_Start_RKNN_SDK`>。

2.编译模型相关文件

```
./build-linux.sh -t rk3588 -a aarch64 -d mobilenet
```

3.推送可执行文件到板端

```
adb root
adb remount
adb push install/rk3588_linux_aarch64/rknn_mobilenet_demo/ /userdata/
```

4.板端执行

```
adb shell
cd /userdata/rknn_mobilenet_demo/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/mobilenet_v2.rknn model/bell.jpg
```

输出结果如下：

```
-----TOP 5-----
[494] score=0.99 class="n03017168 chime, bell, gong"
[469] score=0.00 class="n02939185 caldron, cauldron"
[653] score=0.00 class="n03764736 milk can"
[747] score=0.00 class="n04023962 punching bag, punch bag"
[505] score=0.00 class="n03063689 coffeepot"
```

4.2 YOLOv5模型部署示例

4.2.1 模型转换

1.下载模型

```
cd rknn_model_zoo/examples/yolov5/model
./download_model.sh
```

2.执行模型转换

```
cd rknn_model_zoo/examples/yolov5/python
python convert.py ../model/yolov5s_relu.onnx rk3588 i8 ../model/yolov5s_relu.rknn
```

转换后的模型保存路径为`rknn_model_zoo/examples/yolov5/model/yolov5s_relu.rknn`。

4.2.2 模型连板运行

1.进入rknn_model_zoo/examples/yolov5/python 目录

```
cd rknn_model_zoo/examples/yolov5/python
```

2.执行模型连板运行

```
python yolov5.py --model_path ../model/yolov5s_relu.rknn --target rk3588 --img_show
```

默认输入图片是`model/bus.jpg`，结果图片如下所示：

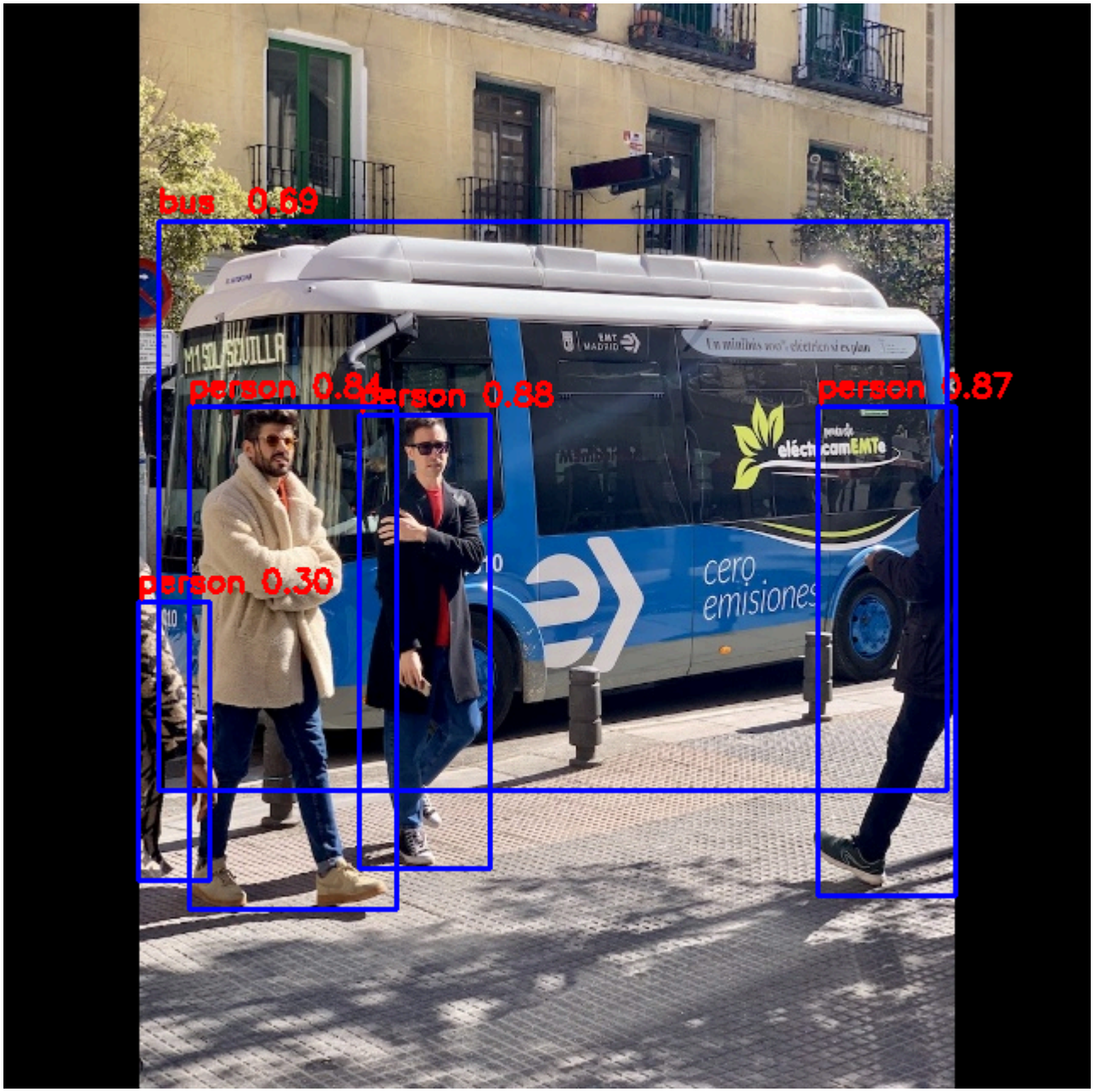


图4-1 RKNN Python Demo可视化结果

4.2.3 板端部署运行

1.在rknn_model_zoo工程下的build-linsx.sh 脚本中指定gcc交叉编译器路径

```
GCC_COMPILER=~/opts/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu
```

有关gcc交叉编译器的下载和安装方法可参考Rockchip_RKNPU_Quick_Start_RKNN_SDK。

2.编译模型相关文件

```
cd rknn_model_zoo
./build-linux.sh -t rk3588 -a aarch64 -d yolov5
```

3.推送可执行文件到板端

```
adb root
adb remount
adb push install/rk3588_linux_aarch64/rknn_yolov5_demo/ /userdata/
adb push examples/yolov5/model/yolov5s_relu.rknn /userdata/rknn_yolov5_demo/model/
```

4.板端运行

```
adb shell
cd userdata/rknn_yolov5_demo/
export LD_LIBRARY_PATH=./lib
./rknn_yolov5_demo model/yolov5s_relu.rknn model/bus.jpg
```

5.从板端拉取到本地查看，在本地电脑的终端中，执行以下命令：

```
adb pull /userdata/rknn_yolov5_demo/out.png .
```

输出结果图片如下所示：

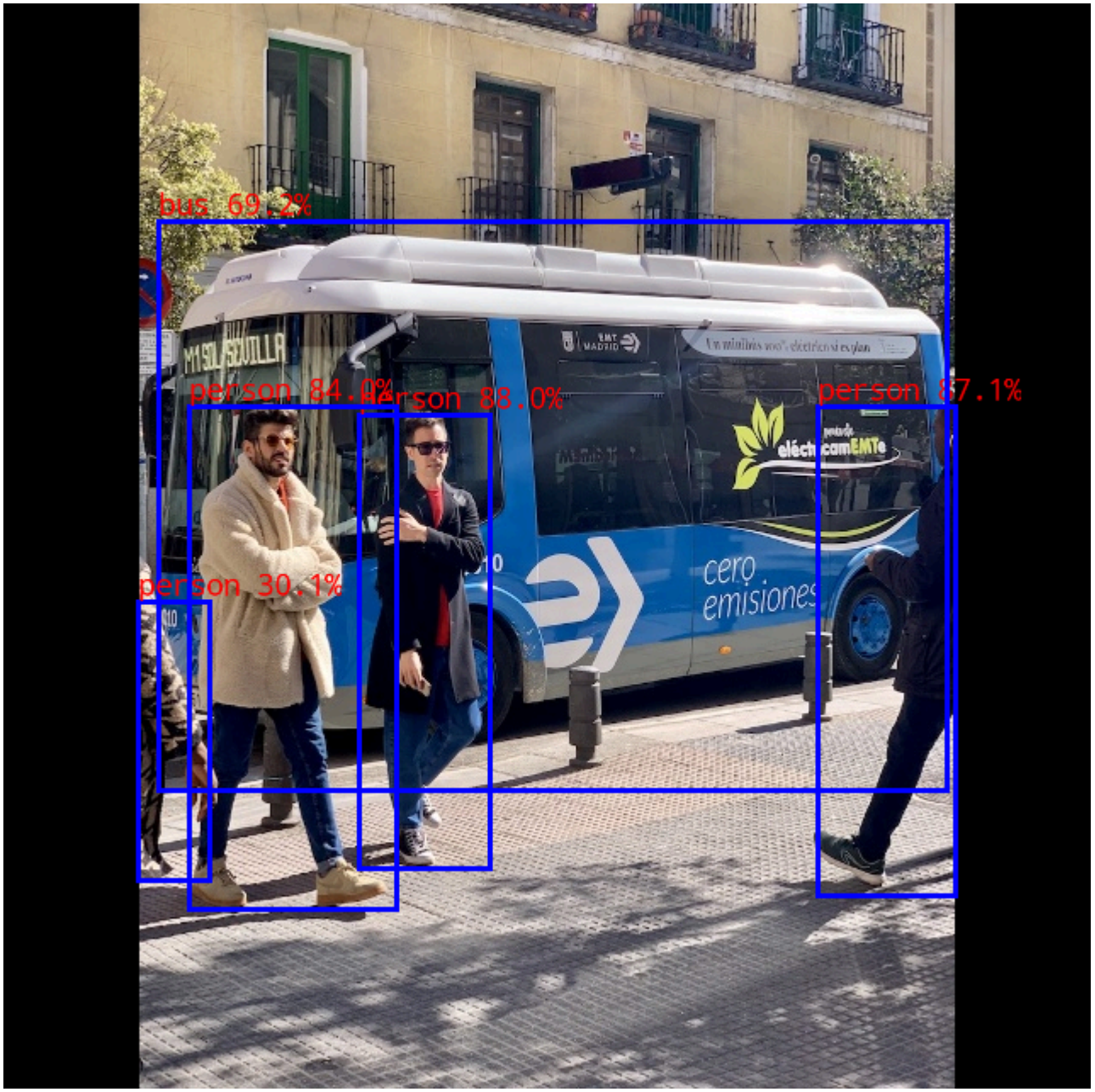


图4-2 RKNN C demo可视化结果

5 RKNN进阶使用说明

5.1 数据排列格式

目前RKNN的数据排列格式主要有以下四种，`NHWC`、`NCHW`、`NC1HWC2`、`UNDEFINE`。

其中`NHWC`和`NCHW`的数据排布为深度学习常见数据排列方式，本章节不做额外说明，重点讲述RKNPU硬件专用的`NC1HWC2`数据格式的存储以及转换。

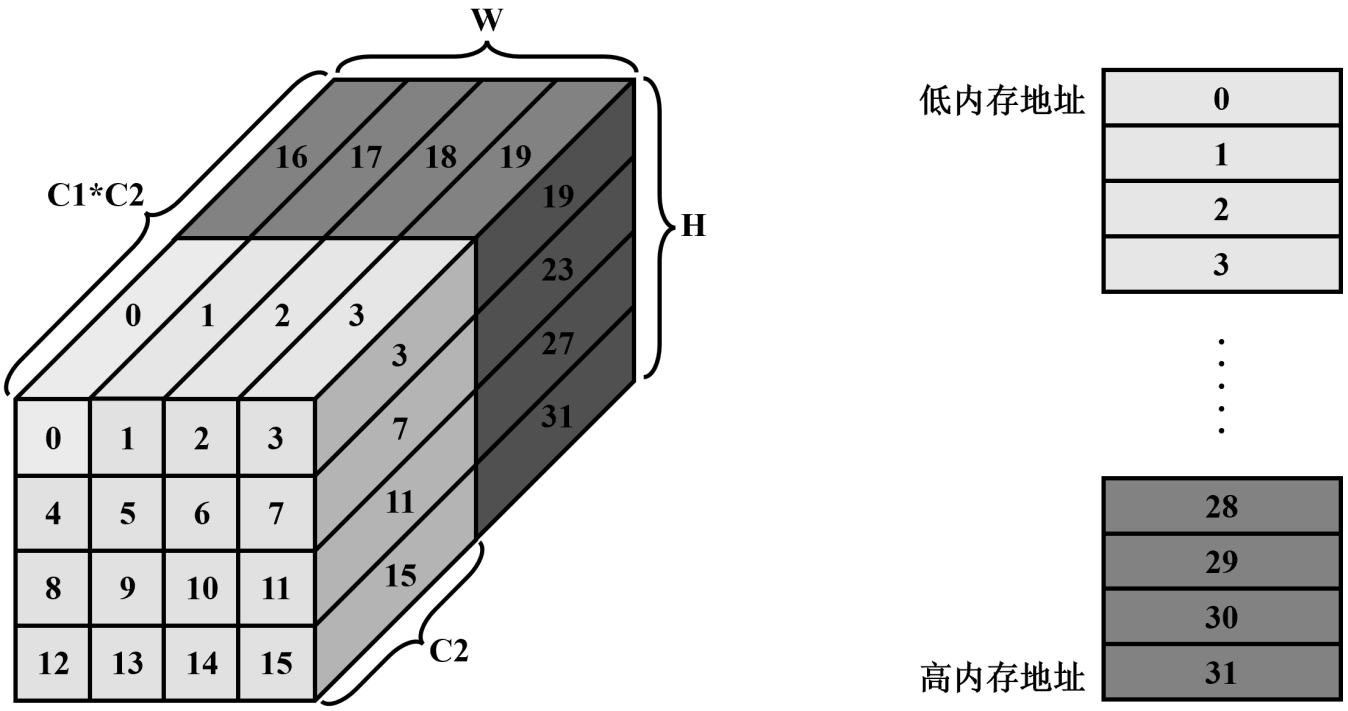


图5-1 RKNPU NC1HWC2数据排布与存储

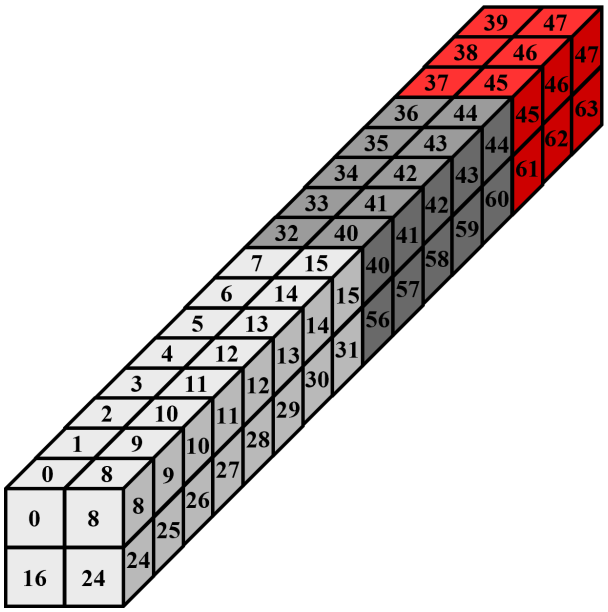
如图5-1所示， 数字0代表一笔数据，即一次存放C2个数据，其中C2是由平台决定的，不同硬件平台的C2的规则约束由表5-1所示，C1为C/C2的上取整值。`NC1HWC2`数据存放的顺序与图中数值增长的顺序一致，先存放0-15的数据，再存放16-31的数据。以RK3568平台为例，当feature为(1,13,4,4)的int8数据，对应的`NC1HWC2`为(1,2,4,4,8),此时C2为8，C1为2，feature在内存中在16-31排放的数据中，对应的每个C2数据块只有前5个数据有效，剩下的3个数据是额外补的对齐数据。

表5-1 不同硬件平台对应的C2值

	RK2118	RK3566/RK3568	RK3588/RK3576/RV1126B	RV1103B/RV1106B	RV1103/RV1106	RK3562
int8	/	8	16	8	16	16
float16	4	4	8	/	8	8

接下来重点介绍`NC1HWC2`数据排列转`NCHW`和`NHWC`数据在内存中的变化过程。

以feature (1, 13, 2, 2) RK3568为例，数据在内存排布中的转换，根据前文的对齐要求可知feature(1, 13, 2, 2) 对应的`NC1HWC2`为(1, 2, 2, 2,8) ，`NC1HWC2`的存储如下图所示，红色部分为额外对齐的无效数据。



低地址

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

高地址

图5-2 NC1HWC2数据排布展开

移除无效数据转成`NCHW`即 (1, 13, 2, 2)数据，在内存中的排布如下：

低地址

0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31
32	40	48	56
33	41	49	57
34	42	50	58
35	43	51	59
36	44	52	60

高地址

图5-3 NCHW 数据排布

移除无效数据转成`NHWC`即 (1, 2, 2, 13)数据，在内存中的排布如下：

低地址	0	1	2	3	4	5	6	7	32	33	34	35	36
	8	9	10	11	12	13	14	15	40	41	42	43	44
	16	17	18	19	20	21	22	23	48	49	50	51	52
高地址	24	25	26	27	28	29	30	31	56	57	58	59	60

图5-4 NHWC 数据排布

转换示例代码：

`NC1HWC2`转`NCHW`：以int8数据排列的`NC1HWC2`转成int8数据排列的`NCHW`如下所示：

```
/*
 *src:     表示NC1HWC2输入tensor的地址
 *dst:     表示NCHW输出tensor的地址
 *dims:    表示NC1HWC2的shape信息
 *channel: 表示NCHW输入的C的值
 * h :     表示NCHW的h的值
 * w:      表示NCHW的w的值
 */
int NC1HWC2_to_NCHW(const int8_t* src, int8_t* dst, int* dims, int channel, int h, int w)
{
  int batch  = dims[0];
  int C1     = dims[1];
  int C2     = dims[4];
  int hw_src = dims[2] * dims[3];
  int hw_dst = h * w;
  for (int i = 0; i < batch; i++) {
    src = src + i * C1 * hw_src * C2;
    dst = dst + i * channel * hw_dst;
    for (int c = 0; c < channel; ++c) {
      int plane  = c / C2;
      const int8_t* src_c  = plane * hw_src * C2 + src;
      int  offset = c % C2;
      for (int cur_h = 0; cur_h < h; ++cur_h)
        for (int cur_w = 0; cur_w < w; ++cur_w) {
          int cur_hw = cur_h * w + cur_w;
          dst[c * hw_dst + cur_h * w + cur_w] = src_c[C2 * cur_hw + offset];
        }
    }
  }
  return 0;
}
```

`NC1HWC2`转`NHWC`：以int8数据排列的`NC1HWC2`转成int8数据排列的`NHWC`如下所示：

```
/*
 *src:     表示NC1HWC2输入tensor的地址
 *dst:     表示NCHW输出tensor的地址
 *dims:    表示NC1HWC2的shape信息
 *channel: 表示NHWC输入的C的值
```

```
 * h :      表示NCHW的h的值
 * w:      表示NCHW的w的值
 */
int NC1HWC2_to_NHWC(const int8_t* src, int8_t* dst, int* dims, int channel, int h, int w)
{
  int batch  = dims[0];
  int C1     = dims[1];
  int C2     = dims[4];
  int hw_src = dims[2] * dims[3];
  int hw_dst = h * w;
  for (int i = 0; i < batch; i++) {
    src = src + i * C1 * hw_src * C2;
    dst = dst + i * channel * hw_dst;
    for (int cur_h = 0; cur_h < h; ++cur_h) {
      for (int cur_w = 0; cur_w < w; ++cur_w) {
        int cur_hw = cur_h * dims[3] + cur_w;
        for (int c = 0; c < channel; ++c) {
          int  plane  = c / C2;
          const auto* src_c   = plane * hw_src * C2 + src;
          int  offset  = c % C2;
          dst[cur_h * w * channel + cur_w * channel + c] = src_c[C2 * cur_hw + offset];
        }
      }
    }
  }
  return 0;
}
```

5.2 RKNN Runtime零拷贝调用

5.2.1 零拷贝介绍

目前在RK3562 / RK3566 / RK3568 / RK3576 / RK3588 / RV1126B上有两组API可以使用，分别是通用API接口和零拷贝流程的API接口，RV1103系列/RV1106系列支持零拷贝流程的API接口。零拷贝示例代码可以参考rknpu2工程下的examples/rknn_zero_copy目录，有关rknpu2的详细内容和代码库请访问链接：https://github.com/airockchip/rknn-toolkit2/tree/master/rknpu2。

在推理RKNN模型时，原始数据要经过输入处理、NPU运行模型、输出处理三大流程。目前根据不同模型输入格式和量化方式，接口内部会存在通用API和零拷贝API两种处理流程，如图5-5和图5-6所示，两组API的主要区别在于，通用接口每次更新帧数据，需要将外部模块分配的数据拷贝到NPU运行时的输入内存，而零拷贝流程的接口会直接使用预先分配的内存（包括NPU运行时创建的或外部其他框架创建的，比如DRM框架），减少了内存拷贝的花销，性能更优，带宽更少。当用户输入数据只有虚拟地址时，只能使用通用API接口；当用户输入数据有物理地址或fd时，两组接口都可以使用。**通用API和零拷贝API不能混合调用。**

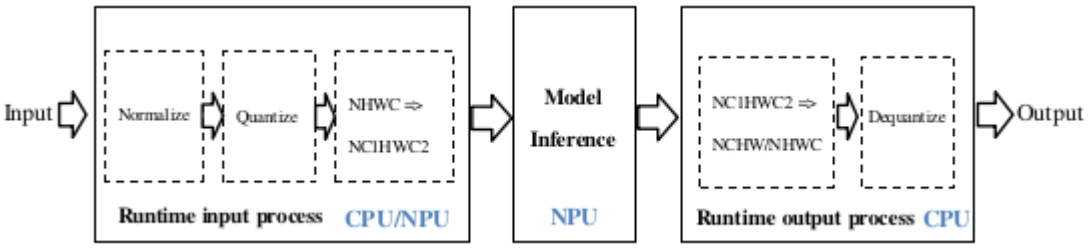


图5-5 通用API的数据处理流程

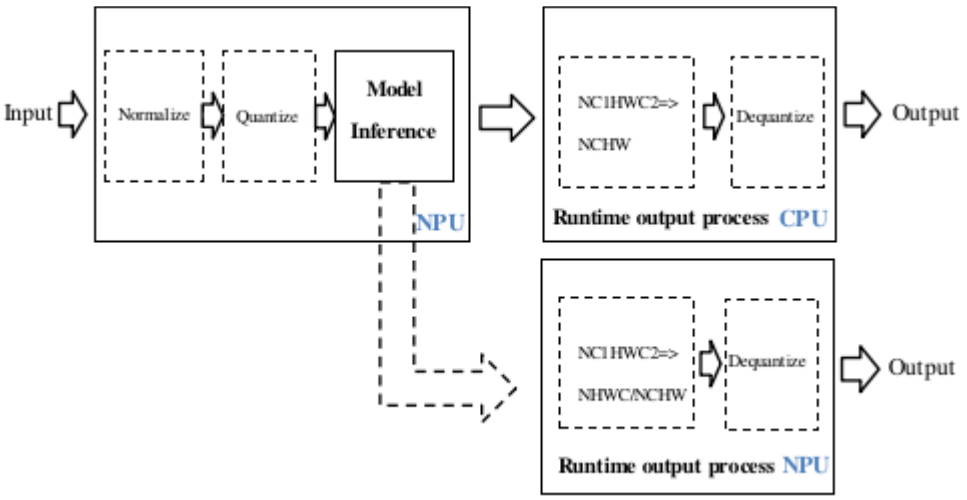


图5-6 零拷贝API数据处理流程

1.通用API

通用API的流程如图5-5所示。对于数据的归一化、量化、数据排布格式转换、反量化在CPU上运行（在符合零拷贝输入要求的情况下，归一化和量化会运行在NPU上，但输入数据仍需要用CPU拷贝到模型的输入buffer上），模型本身的推理在NPU上运行。

2.零拷贝API

零拷贝API的流程如图5-6所示。优化了通用API的数据处理流程，零拷贝API归一化、量化和模型推理都会在NPU上运行，NPU输出的数据排布格式和反量化过程在CPU或者NPU上运行。零拷贝API对于输入数据流程的处理效率会比通用API高。

零拷贝场景的使用条件如下表所示：

表5-2 零拷贝输入要求

输入维度	输入对齐要求	
	RV1103B / RV1106B / RK3566 / RK3568	RK3562 / RK3576 / RK3588 / RV1103 / RV1106 / RV1126B
4维，通道数是1、3、4	宽8字节对齐	宽16字节对齐
非4维	总大小8字节对齐	总大小16字节对齐

5.2.2 C API零拷贝整体流程

零拷贝API接口使用`rknn_tensor_memory`结构体，需要在推理前创建并设置该结构体，并在推理后读取该结构体中的内存信息。根据用户是否需要自行分配模型的模块内存（输入/输出/权重/中间结果）和内存表示方式（文件描述符/物理地址等）差异，有下列三种典型的零拷贝调用流程，如图5-7至图5-9所示，红色部分表示专为零拷贝加入的接口和数据结构，斜体表示接口调用之间传递的数据结构。

- 输入/输出内存由运行时分配

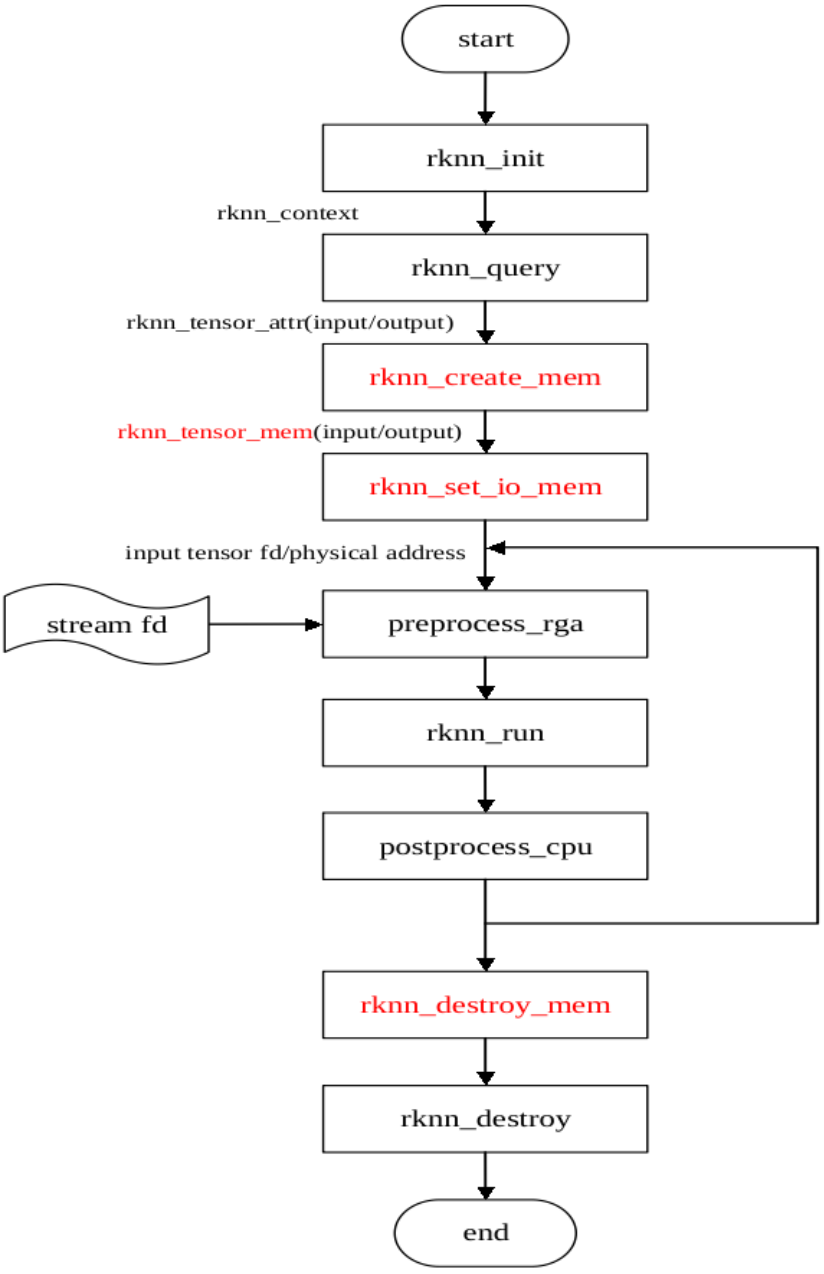


图5-7 零拷贝 API接口调用流程（输入/输出内部分配）

如图5-7所示，输入/输出内存由运行时分配调用的是`rknn_create_mem()`接口创建`rknn_tensor_memory`结构体，`rknn_set_io_mem()`设置输入输出`rknn_tensor_memory`结构体。

`rknn_create_mem()`接口创建的输入/输出内存信息结构体包含了文件描述符成员和物理地址，RGA的接口使用到NPU分配的内存信息， `preprocess_rga()`表示RGA的接口， stream_fd表示RGA的接口输入源的内存数据，`postprocess_cpu()`表示后处理的CPU实现。

- 输入/输出内存由外部分配

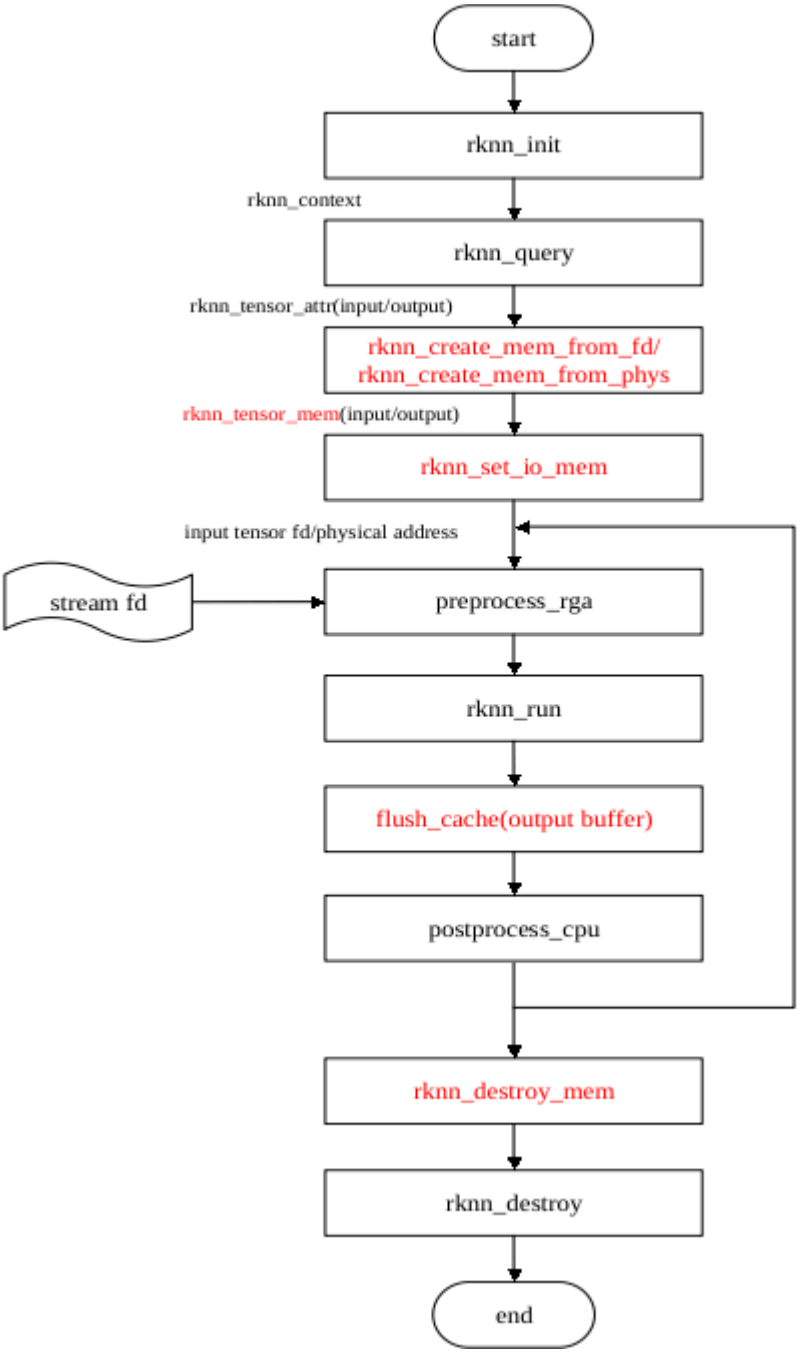


图5-8 零拷贝API接口调用流程（输入/输出外部分配）

如图5-8所示，输入/输出内存由外部分配调用的是`rknn_create_mem_from_fd()`/`rknn_create_mem_from_phys()`接口创建`rknn_tensor_memory`结构体，`rknn_set_io_mem()`设置输入输出`rknn_tensor_memory`结构体。`flush_cache`表示用户需要调用与分配的内存类型关联的接口来刷新输出缓存。

- 输入/输出/权重/中间结果内存由外部分配

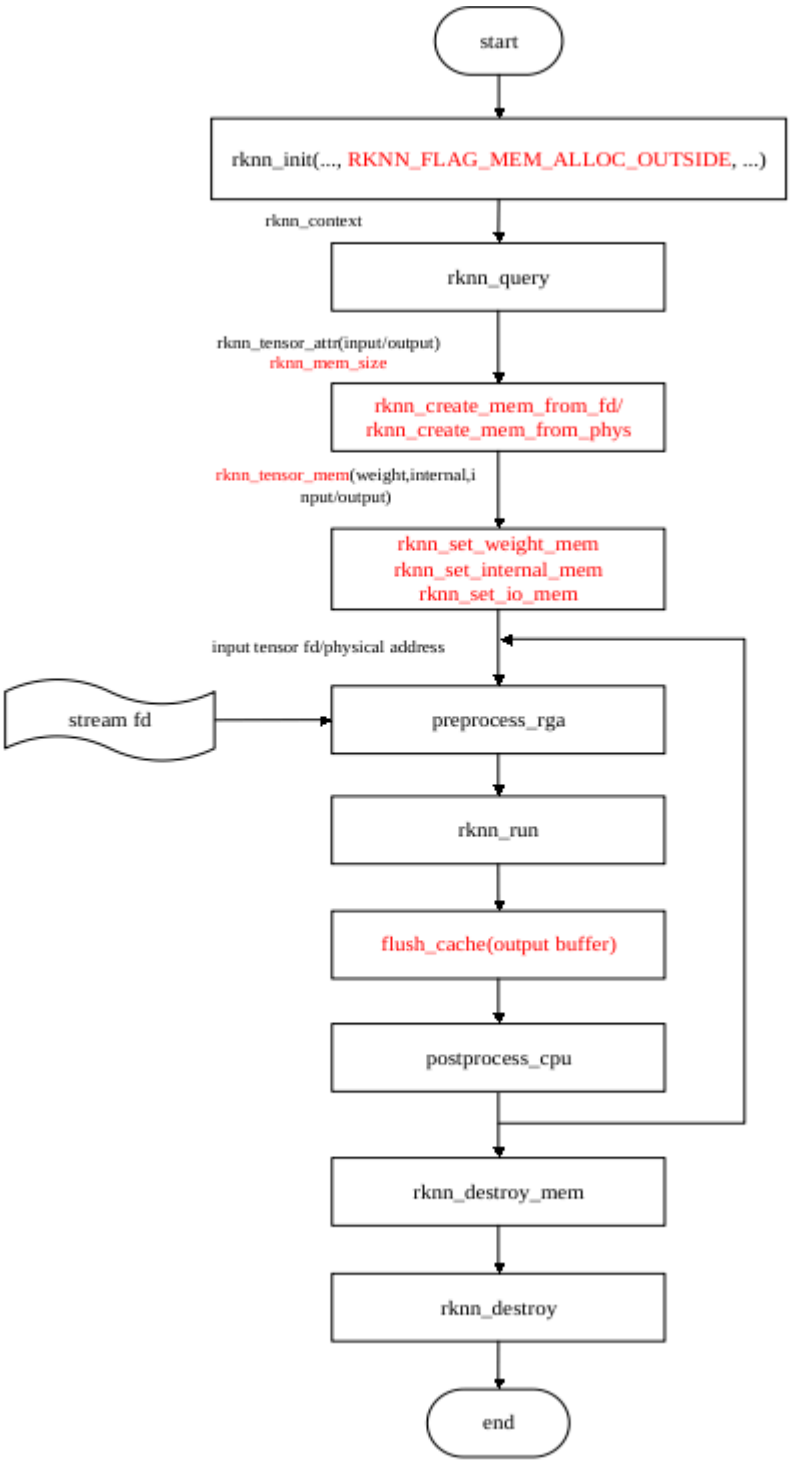


图5-9 零拷贝 API接口调用流程（输入/输出/权重/中间结果外部分配）

如图5-9所示，输入/输出/权重/中间结果内存由外部分配调用的是`rknn_create_mem_from_fd()`/`rknn_create_mem_from_phys()`接口创建`rknn_tensor_memory`结构体，`rknn_set_io_mem()`设置输入输出`rknn_tensor_memory`结构体，`rknn_set_weight_mem()`/`rknn_set_internal_mem()`设置权重/中间结果`rknn_tensor_memory`结构体。

5.2.3 C API零拷贝的用法

以图5-7零拷贝API接口调用流程（输入/输出内部分配）为例，用法如下：

- rknn_query()

输入:

用**RKNN_QUERY_NATIVE_INPUT_ATTR**查询相关的属性（注意，不是**RKNN_QUERY_INPUT_ATTR**）. 当查询出来的fmt（或者称为layout）不同时，需要提前处理的方式也不一样。该方式查询出来的是输入硬件效率最优的layout和type。

`rknn_query()`输入的情况如下:

a. 当layout为`RKNN_TENSOR_NCHW`时，这种情况一般输入是4维，并且数据类型为bool或者int64，当传数据给NPU时，也需要按照NCHW格式排列给NPU。

b. 当layout为`RKNN_TENSOR_NHWC`时，这种情况一般输入是4维，并且数据类型为float32/float16/int8/uint8，同时，输入通道数是1、3、4。当传数据给NPU时，也需要按照`NHWC`格式排列给NPU。需要注意的是当`pass_through=1`时，width可能需要做stride对齐，具体取决于查询出来的`w_stride`的值。

c. 当layout为`RKNN_TENSOR_NC1HWC2`时，这种情况一般输入是4维，并且数据类型为float16/int8，同时，输入通道数不是1、3、4。当`pass_through=0`时，输入数据按照`NHWC`格式排列，接口内部会进行`NHWC`到`NC1HWC2`的 CPU 转换; 当`pass_through=1`时，输入数据按照`NC1HWC2`格式排列, 用户外部需转换好。

d. 当layout为`RKNN_TENSOR_UNDEFINED`时，这种情况一般输入不是4维，当传数据给NPU时，需要按照ONNX模型输入格式传给NPU。NPU不做任何的mean/std处理以及layout转换。

如果用户需要的输入配置不同于查询接口获取的`rknn_tensor_attr`结构体，可以对`rknn_tensor_attr`结构体进行对应修改，目前支持的可修改的输入数据类型如表5-3所示.特别注意：如果查询的数据类型是uint8，用户想传入float32类型，则`rknn_tensor_attr`结构体的size要修改成原size的四倍，同时其中的数据类型要修改成`RKNN_TENSOR_FLOAT32`。用该方式修改后硬件效率就不是最优了，接口内部会调用CPU进行数据类型转换。

表5-3 输入可修改的输入数据类型表

	rknn_query 得到的模型数据类型						
用户接口修改的数据类型		bool	int8	float16	int16	int32	Int64
	bool	Y					
	int8		Y				
	uint8		Y	Y			
	float32		Y	Y			
	float16			Y			
	int16				Y		
	int32					Y	
	Int64						Y

输出:

用**RKNN_QUERY_NATIVE_OUTPUT_ATTR**查询相关的属性（注意，不是**RKNN_QUERY_OUTPUT_ATTR**）. 当查询出来的fmt（或者称为layout）不同时，需要后处理的方式也不一样。该方式查询出来的是输出硬件效率最优的layout和type。

当输出是4维且用户需要`NHWC` layout的四维输出时, 可以用`RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR`查询相关的属性。该方式可以直接获得NHWC layout的输出。

`rknn_query()`输出的情况如下:

a. 当layout为`RKNN_TENSOR_NC1HWC2`时，这种情况一般输出是4维，并且数据类型为float16/int8。当用户需要`NCHW` layout时，外部需进行`NC1HWC2`到`NCHW`的layout 转换。

b. 当layout为`RKNN_TENSOR_UNDEFINED`时，这种情况一般输出非4维，并且数据类型为float16/int8。用户外部无需进行layout转换。

c. 当layout为`RKNN_TENSOR_NCHW`时，这种情况一般输出是4维, 并且数据类型为float16/int8。用户外部无需进行layout转换。

d. 当layout为`RKNN_TENSOR_NHWC`时，这种情况一般输出是4维, 并且数据类型为float16/int8。这种情况一般是用户调用`RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR`接口查询出来的layout。

如果用户需要的输出配置不同于查询接口获取的`rknn_tensor_attr`结构体，可以对`rknn_tensor_attr`结构体进行对应修改，可修改的配置信息如表5-4，表5-5所示，特别注意：如果查询输出的数据类型是int8，用户想获取成float32类型输出，则`rknn_tensor_attr`结构体的size要修改成原size的四倍，同时其中的数据类型要修改成`RKNN_TENSOR_FLOAT32`。用该方式修改后硬件效率就不是最优了，接口内部会调用CPU进行数据类型转换。

表5-4 输出可修改的输入数据类型表

	rknn_query 得到的模型数据类型						
用户接口修改的数据类型		bool	int8	float16	int16	int32	Int64
	bool	Y					
	int8		Y				
	uint8						
	float32		Y	Y			
	float16			Y			
	int16				Y		
	int32					Y	
	Int64						Y

表5-5 输出可修改的layout类型表

	rknn_query查询得到的模型layout类型				
		NC1HWC2	NCHW	NHWC	UNDEFINE
用户接口设置的layout类型	NC1HWC2	Y			
	NCHW	Y	Y		
	NHWC			Y	
	UNDEFINE				Y

RK3562/RK3566/RK3568/RK3576/RK3588/RV1126B支持的零拷贝接口NPU输出配置如表5-6所示，RV1106/RV1106B/RV1103/RV1103B支持零拷贝接口NPU输出配置如表5-7所示。

表5-6 RK3562/RK3566/RK3568/RK3576/RK3588/RV1126B零拷贝接口NPU支持的输出配置

模型类型	输出数据类型	输出维度	可支持output layout
int8模型	int8/float16/float32	4维	NCHW/NC1HWC2/NHWC
		非4维	UNDEFINE
float16模型	float16/float32	4维	NCHW/NC1HWC2/NHWC
		非4维	UNDEFINE

表5-7 RV1103/RV1103B/RV1106/RV1106B零拷贝接口NPU支持的输出配置

模型类型	输出数据类型	输出维度	可支持output layout
int8 模型	int8/float16	4维	NCHW/NC1HWC2/NHWC
		非4维	UNDEFINE

- rknn_create_mem

零拷贝API接口使用`rknn_tensor_memory`结构体，需要在推理前创建并设置该结构体，并在推理后读取该结构体中的内存信息。当无需对`RKNN_QUERY_NATIVE_INPUT_ATTR`，`RKNN_QUERY_NATIVE_OUTPUT_ATTR`查询出来的layout和type进行修改时，直接采用默认配置的`size_with_stride`创建内存大小。若修改了相应的layout和type，则需按照相应的size 创建内存大小.(例如输出的数据类型是int8，用户想获取成float32类型输出，size要修改成原size的四倍)。

- rknn_set_io_mem

`rknn_set_io_mem()`用于设置包含模型输入/输出内存信息的`rknn_tensor_mem`结构体，和`rknn_init()`类似，只要在最开始调用一次，后面反复执行`rknn_run()`即可。

5.3 NPU多核配置

RK3588通过3核NPU，RK3576通过2核NPU提供更强的算力。本章节将详细介绍多核NPU的配置方法，以提高模型的推理效率。

注：多核运行适用于网络层计算量较大的网络，对小网络提升幅度较小，甚至可能因为单核多核的切换（该切换需CPU介入）而导致性能下降。

5.3.1 多核运行配置方法

如果使用Python作为应用程序开发语言，可以通过RKNN-Toolkit2或RKNN-Toolkit Lite2 `init_runtime()`接口中的“`core_mask`”参数设置模型运行的 NPU 核心。该参数的详细说明如下表：

表5-8 init_runtime接口core_mask参数说明

参数	详细说明
core_mask	该参数用于设置模型运行的NPU核心。可选值和相应说明如下： NPU_CORE_AUTO: 自动调度模式，模型将以单核模式自动运行在当前空闲的NPU核上。 NPU_CORE_0: 模型运行在NPU Core0上。 NPU_CORE_1: 模型运行在NPU Core1上。 NPU_CORE_2: 模型运行在NPU Core2上。 NPU_CORE_0_1: 模型同时运行在NPU Core0和NPU Core1上。 NPU_CORE_0_1_2: 模型同时运行在NPU Core0，Core1和Core2上。 NPU_CORE_ALL: 根据平台自动配置NPU核心数量。 默认值为 NPU_CORE_AUTO。 注：在RKNN-Toolkit Lite2上设置该参数时，值的前面要加上RKNNLite，例如RKNNLite.NPU_CORE_AUTO；如果在 RKNN-Toolkit2 上设置该参数时，值的前面要加上RKNN，例如RKNN.NPU_CORE_AUTO。

RKNN-Toolkit2设置NPU核心，参考代码如下：

```
## Python
......
# Init runtime environment
print('--> Init runtime environment')
ret = rknn.init_runtime(target='rk3588', core_mask=RKNN.NPU_CORE_0)
if ret != 0:
    print('Init runtime environment failed!')
    exit(ret)
```

RKNN-Toolkit Lite2设置NPU核心，参考代码如下：

```
## Python
......
# Init runtime environment
print('--> Init runtime environment')
ret = rknn_lite.init_runtime(core_mask=RKNNLite.NPU_CORE_0)
if ret != 0:
    print('Init runtime environment failed')
    exit(ret)
print('done')
```

如果使用 C/C++ 作为应用程序开发语言，可以调用`rknn_set_core_mask()`接口设置模型运行的NPU核心。该接口`core_mask`参数的详细说明如下表：

表5-9 rknn_set_core_mask接口core_mask参数说明

参数	详细说明
core_mask	该参数用于设置模型运行的NPU核心。可选值和相应说明如下： RKNN_NPU_CORE_AUTO: 自动调度模式，模型将以单核模式自动运行在当前空闲的NPU核上。 RKNN_NPU_CORE_0: 模型运行在NPU Core0上。 RKNN_NPU_CORE_1: 模型运行在NPU Core1上。 RKNN_NPU_CORE_2: 模型运行在NPU Core2上。 RKNN_NPU_CORE_0_1: 模型同时运行在NPU Core0和NPU Core1上。 RKNN_NPU_CORE_0_1_2: 模型同时运行在NPU Core0，Core1和Core2上。 RKNN_NPU_CORE_ALL: 根据平台自动配置NPU核心数量。

使用C/C++ API设置模型运行NPU核心，参考代码如下：

```
// C++
// rknn_context ctx;
rknn_core_mask core_mask = RKNN_NPU_CORE_0;
int ret = rknn_set_core_mask(ctx, core_mask);
```

5.3.2 查看多核运行效果

本章节将详细说明RKNN模型以多核模式运行时的效果。

如果使用RKNN-Toolkit2连接开发板进行模型推理，需要在调用`rknn.init_runtime()`接口时将`perf_debug`参数设置成`True`，接着调用`rknn.eval_perf()`接口，即可打印每层的运行信息。参考代码如下：

```
#Python
# Init runtime environment
ret = rknn.init_runtime(target='rk3588', device_id='29d5dd97766a5c27', perf_debug=True)
if ret != 0:
    print('Init runtime environment failed!')
    exit(ret)

# Eval performance
rknn.eval_perf()
```

如果是直接在板端进行模型推理，需要在运行应用前将RKNN_LOG_LEVEL设成4或以上，此时将打印模型每层的运行信息。设置方法如下：

```
# Python
# 使用RKNN-Toolkit Lite2提供的Python接口，只需在创建RKNNLite对象时将verbose设成True即可
rknnlite = RKNNLite(verbose=True)

# 使用C/C++接口，则需要在运行二进制程序前设置如下环境变量
export RKNN_LOG_LEVEL=4
```

以lenet模型为例，通过上述设置后，终端将打印类似如下日志（为方便展示，删除了InputShape, OutputShape, DDR Cycles, NPU Cycles, Total Cycles, Time(us), MacUsage(%), Task Number, Lut Number, RW(kb), FullName等字段）：

```
ID   OpType           DataType Target WorkLoad(0/1/2)-ImproveTherical
1    InputOperator    UINT8    CPU    100.0%/0.0%/0.0% - Up:0.0%
2    Conv               UINT8    NPU    50.0%/50.0%/0.0% - Up:50.0%
3    MaxPool            INT8    NPU    100.0%/0.0%/0.0% - Up:0.0%
4    Conv                INT8    NPU    50.0%/50.0%/0.0% - Up:50.0%
5    MaxPool            INT8    NPU    100.0%/0.0%/0.0% - Up:0.0%
6    ConvRelu           INT8    NPU    48.1%/51.9%/0.0% - Up:48.1%
7    Conv                INT8    NPU    100.0%/0.0%/0.0% - Up:0.0%
8    Softmax            INT8    CPU    0.0%/0.0%/0.0% - Up:0.0%
9    OutputOperator   FLOAT16 CPU    0.0%/0.0%/0.0% - Up:0.0%
Total Operator Elapsed Time(us): 591
Total Memory RW Amount(MB): 0
```

模型每层运行信息中的"`WorkLoad(0/1/2)-ImproveTherical`"一列只在多核NPU上会打印，记录了模型每一层的任务在NPU核心上是如何分配以及其理论性能提升情况。例如"`50.0%/50.0%/0.0% - Up:50.0%`"代表该层的计算量以Core0负责50%，Core1负责50%进行分配，该层的性能相比单核运行，理论能提升50%。如果某一层的性能没有提升，例如"100.0%/0.0%/0.0% - Up:0.0%"，可能存在以下几种情况：

该层的负载太小，小于NPU多核任务分配的粒度，因此该层运行在单核上；

该类算子在NPU驱动中未实现多核任务切分，待后续版本支持。现有已支持多核任务切分的算子有：**Conv, DepthwiseConvolution, Add, Concat, Relu, Clip, Relu6, ThresholdedRelu, PRelu, LeakyRelu**。

5.3.3 多核性能提升技巧

可以尝试如下方法，以得到较高的多核运行性能：

- 将CPU/DDR/NPU频率定到最高
- 将应用绑定至CPU大核
- 将NPU中断绑定至应用所对应的CPU大核

不同固件对应的定频命令有所区别，请参考8.1.1章节。

以将应用绑定到CPU4大核心为例，上面提到的后两点可以参考如下脚本：

```
interrupts=$(cat /proc/interrupts | grep npu)
interrupts_array=($interrupts)

irq1=$(echo ${interrupts_array[0]} | awk -F ':' '{print $1}')
irq2=$(echo ${interrupts_array[14]} | awk -F ':' '{print $1}')
irq3=$(echo ${interrupts_array[28]} | awk -F ':' '{print $1}')

for irq in $irq1 $irq2 $irq3; do
  echo 4 > /proc/irq/$irq/smp_affinity_list
done

taskset 10 ./rknn_benchmark lenet.rknn "" 10 3 ## CPU4对应的 taskset 掩码值为 0x10
```

上述脚本会执行如下操作：

- 执行`cat /proc/interrupts | grep npu`命令并解析出三个中断号（去除冒号）
- 使用循环将每个中断号的`smp_affinity_list`设置为4（CPU4对应的ID为4）
- 最后执行t`askset 10 ./rknn_benchmark lenet.rknn "" 10 3`命令，CPU4对应的`taskset`参数为10（有关`taskset`的具体用法，请参考：https://man7.org/linux/man-pages/man1/taskset.1.html）

通过上述操作，NPU中断以及应用程序"`rknn_benchmark`"都将在CPU4上运行，这样可以消除NPU中断处理的核心切换开销。

5.4 动态Shape

5.4.1 动态Shape功能介绍

动态shape是指模型输入数据的形状在运行时可以改变。它可以帮助处理输入数据大小不固定的情况，增加模型的灵活性。在之前仅支持静态shape的RKNN模型情况下，如果用户需要使用多个输入shape，传统的做法是生成多个RKNN模型，在模型部署时初始化多个上下文分别执行推理，而在引入动态shape后，用户可以只保留一份与静态shape RKNN模型大小接近的动态shape RKNN模型，并使用一个上下文进行推理，从而节省Flash占用和DDR占用，动态shape在图像处理和序列模型推理中具有重要的作用，它的典型应用场景包括：

- 序列长度改变的模型，常见于NLP模型，例如BERT, GPT
- 空间维度变化的模型，例如分割和风格迁移
- 带Batch模型，Batch维度上变化
- 可变输出数量的目标检测模型

5.4.2 RKNN SDK版本和平台要求

- RKNN-Toolkit2版本>=1.5.0
- RKNPU Runtime库(librknnrt.so)版本>=1.5.0
- RK3562/RK3566/RK3568/RK3576/RK3588/RK3588S/RV1126B平台的NPU支持该功能

5.4.3 生成动态Shape的RKNN模型

本节介绍使用RKNN-Toolkit2的Python接口生成动态shape的RKNN模型的步骤：

1.确认模型支持动态shape

如果模型文件本身不是动态shape, RKNN-Toolkit2支持扩展成动态shape的RKNN模型。首先，用户要确认模型本身不存在限制动态shape的算子或子图结构，例如，常量的形状无法改变，RKNN-Toolkit2工具在转换过程会报错，如果遇到不支持动态shape扩展的情况，用户要根据报错信息，修改模型结构，重新训练模型以支持动态shape。建议使用原始模型本身就是动态shape的模型。

2.设置需要使用的输入形状

由于NPU硬件特性，动态shape RKNN模型不支持输入形状任意改变，要求用户设置有限个输入形状。对于多输入的模型，每个输入的shape个数要相同。例如，在使用RKNN-Toolkit2转换Caffe模型时，Python代码示例如下：

```
# Python
dynamic_input = [
    [[1,3,224,224]],    # set the first shape for all inputs
    [[1,3,192,192]],    # set the second shape for all inputs
    [[1,3,160,160]],    # set the third shape for all inputs
]

# Pre-process config
rknn.config(mean_values=[103.94, 116.78, 123.68], std_values=[58.82, 58.82, 58.82],
quant_img_RGB2BGR=True, dynamic_input=dynamic_shapes)
```

上述接口配置会生成支持3个shape分别是[1,3,224,224]、[1,3,192,192]和[1,3,160,160]的动态shape RKNN模型。

`dynamic_input`中的shape与原始模型框架的layout一致。例如，对于相同的224x224大小的RGB图片做分类，TensorFlow/TFLite模型输入是[1,224,224,3]，而ONNX模型输入是[1,3,224,224]。

3.量化

在设置好输入shape后， 如果要做量化，则需要设置量化矫正集数据。工具会读取用户设置的最大分辨率输入做量化（是所有输入尺寸之和的最大的一组shape）。例如，模型有两个输入，一个输入shape分别是[1,224]和[1,112]，另一个输入shape分别[1,40]和[1,80]，第一组shape所有输入尺寸之和是1*224+1*40=264，第二组shape所有输入尺寸之和是1*112+1*80=192，第一组shape所有输入尺寸之和更大，因此使用两个输入分别以[1,224]和[1,40]的shape做量化。

- 如果量化矫正集是jpg/png图片格式，用户可以使用不同的分辨率的图片做量化，因为工具会对图片使用opencv的resize方法缩放到最大分辨率后做量化。
- 如果量化矫正集是npy格式，则用户必须使用最大分辨率输入的shape。量化后，模型内所有shape在运行时使用同一套量化参数进行推理。

另外，输入的最大分辨率shape在调用rknn.config时也会打印出来，如下：

```
W config: The 'dynamic_input' function has been enabled, the MaxShape is dynamic_input[0] =
[[1,224],[1,40]]!
The following functions are subject to the MaxShape:
1. The quantified dataset needs to be configured according to MaxShape
2. The eval_perf or eval_memory return the results of MaxShape
```

4.推理评估或精度分析

动态shape RKNN模型做推理或做精度分析时，用户必须提供第2步中设置的其中一组shape的输入。接口使用上与静态shape RKNN模型场景一致，此处不做赘述。

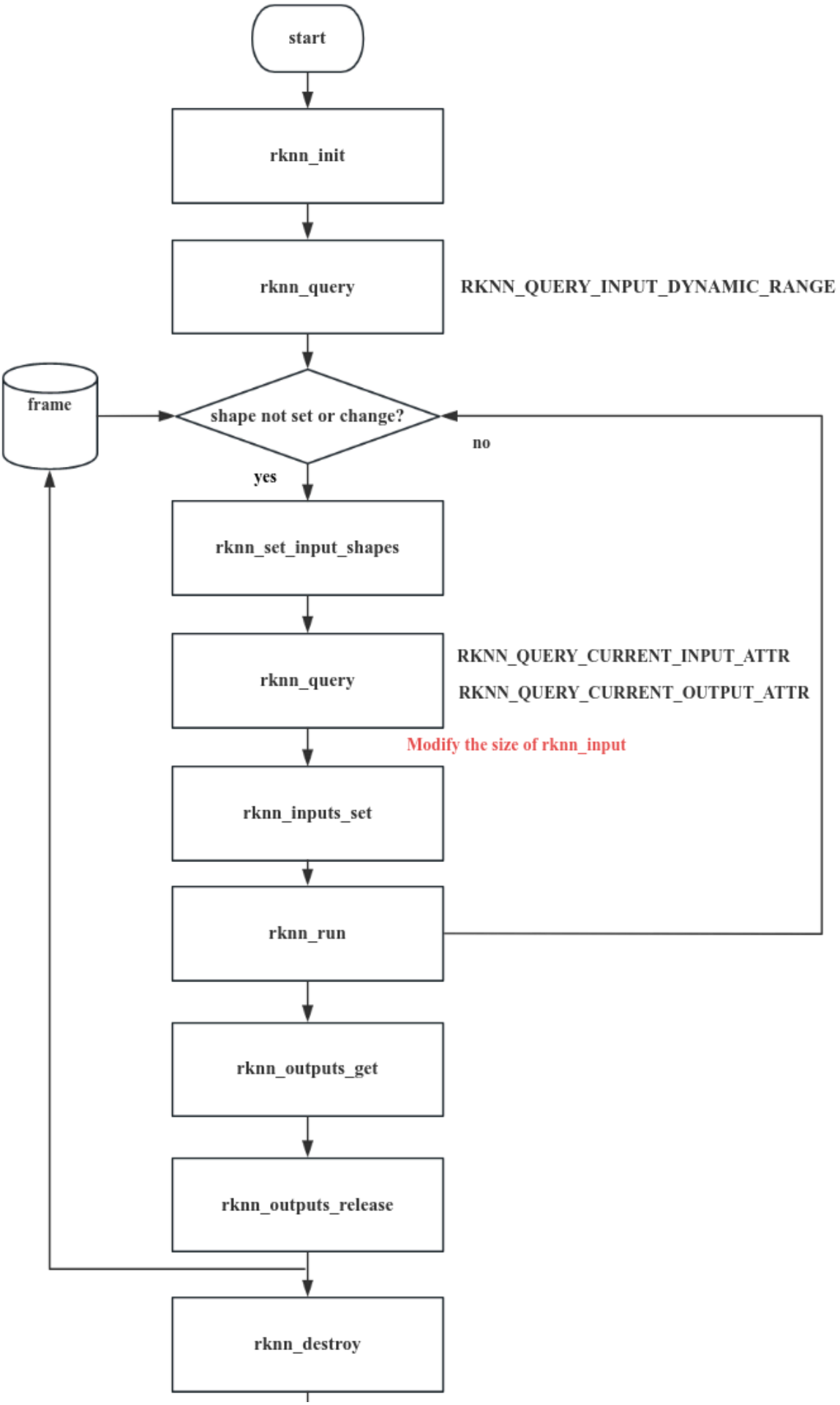
完整的创建动态shape RKNN模型示例，请参考(https://github.com/airockchip/rknn-toolkit2/tree/master/rknn-toolkit2/examples/functions/dynamic_shape)

5.4.4 C API部署

得到动态shape RKNN模型后，接着使用RKNPU2 C API进行部署。按照接口形式，分为通用API和零拷贝API部署流程。

5.4.4.1 通用API

使用通用API部署动态shape RKNN模型的流程如下图所示：


start
rknn_init
rknn_query
RKNN_QUERY_INPUT_DYNAMIC_RANGE
frame
shape not set or change?
no
yes
rknn_set_input_shapes
rknn_query
RKNN_QUERY_CURRENT_INPUT_ATTR
RKNN_QUERY_CURRENT_OUTPUT_ATTR
Modify the size of rknn_input
rknn_inputs_set
rknn_run
rknn_outputs_get
rknn_outputs_release
rknn_destroy

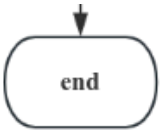


图5-10 动态shape输入接口的通用API调用流程

加载动态shape RKNN模型后，可以在运行时动态修改输入的shape。首先，通过`rknn_query()`可以查询RKNN模型支持的输入shape列表,每个输入支持的shape列表信息以`rknn_input_range`结构体形式返回，它包含了每个输入的名称、数据布局信息、shape个数以及具体shape。接着，通过调用`rknn_set_input_shapes()`接口，传入包含每个输入shape信息的`rknn_tensor_attr`数组指针可以设置当前推理使用的shape。在设置输入shape后，可以再次调用`rknn_query()`查询当前设置成功后的输入和输出shape。

最后，按照通用API流程完成推理。每次切换输入shape时，需要再设置一次新的shape，准备新shape大小的数据并再次调用`rknn_inputs_set()`接口。如果推理前不需要切换输入shape，无需重复调用`rknn_set_input_shapes()`接口。

1.初始化

调用`rknn_init()`接口初始化动态shape RKNN模型，

对于动态shape RKNN模型，在初始化上下文时有如下限制：

- 不支持权重共享功能（带`RKNN_FLAG_SHARE_WEIGHT_MEM`标志的初始化）。
- 不支持上下文复用功能（具体说明见`rknn_dup_context`接口）。

2.查询RKNN模型支持的输入shape组合

初始化成功后，通过`rknn_query()`可以查询到RKNN模型支持的输入shape列表,每个输入支持的shape列表信息以`rknn_input_range`结构体形式返回，它包含了每个输入的名称，layout信息，支持的shape个数以及具体shape。C代码示例如下：

```
// 查询模型支持的输入shape
rknn_input_range dyn_range[io_num.n_input];
memset(dyn_range, 0, io_num.n_input * sizeof(rknn_input_range));
for (uint32_t i = 0; i < io_num.n_input; i++)
{
    dyn_range[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_DYNAMIC_RANGE, &dyn_range[i],
sizeof(rknn_input_range));
    if (ret != RKNN_SUCC)
    {
        fprintf(stderr, "rknn_query error! ret=%d\n", ret);
        return -1;
    }
    dump_input_dynamic_range(&dyn_range[i]);
}
```

注意：对于多输入的模型，所有输入的shape按顺序一一对应，例如，有两个输入、多种shape的RKNN模型，第一个输入的第一个shape与第二个输入的第一个shape组合有效，不存在交叉的shape组合。例如，模型有两个输入A和B，A的shape分别是[1,224]和[1,112]，B的shape分别[1,40]和[1,80]，此时，只支持以下两组输入shape的情况：

- A shape = [1,224],B shape=[1,40]
- A shape = [1,112],B shape=[1,80]

3.设置输入shape

在首次设置输入数据或者输入数据shape发生改变时，需要调用`rknn_set_input_shapes()`接口动态修改输入shape。加载动态shape RKNN模型后，可以在运行时动态修改输入的shape。通过调用`rknn_set_input_shapes()`接口，传入所有输入的`rknn_tensor_attr`数组，每个`rknn_tensor_attr`中的`dims`,`n_dims`和`fmt`三个成员信息表示了当前推理所用的shape。C代码示例如下：

```
/**
dynamic inputs shape range:
index=0, name=data, shape_number=2, range=[[1, 224, 224, 3],[1, 112, 224, 3]], fmt = NHWC
**/
input_attrs[0].dims[0] = 1;
input_attrs[0].dims[1] = 224;
input_attrs[0].dims[2] = 224;
input_attrs[0].dims[3] = 3;
input_attrs[0].fmt=RKNN_TENSOR_NHWC;
ret = rknn_set_input_shapes(ctx, io_num.n_input, input_attrs);
if (ret < 0)
{
    fprintf(stderr, "rknn_set_input_shapes error! ret=%d\n", ret);
    return -1;
}
```

其中，`io_num.n_input`是输入数量，`input_attrs`是模型输入的`rknn_tensor_attr`结构体数组。

注：这里设置的shape必须包含在第2步查询到的shape列表中。

在设置输入shape后，可以再次调用`rknn_query`查询当前设置成功后的输入和输出shape，C代码示例如下：

```
// 获取当前次推理的输入和输出shape
rknn_tensor_attr cur_input_attrs[io_num.n_input];
memset(cur_input_attrs, 0, io_num.n_input * sizeof(rknn_tensor_attr));
for (uint32_t i = 0; i < io_num.n_input; i++)
{
    cur_input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_CURRENT_INPUT_ATTR, &(cur_input_attrs[i]),
sizeof(rknn_tensor_attr));
    if (ret < 0)
    {
        printf("rknn_init error! ret=%d\n", ret);
        return -1;
    }
    dump_tensor_attr(&cur_input_attrs[i]);
}
rknn_tensor_attr cur_output_attrs[io_num.n_output];
memset(cur_output_attrs, 0, io_num.n_output * sizeof(rknn_tensor_attr));
for (uint32_t i = 0; i < io_num.n_output; i++)
{
    cur_output_attrs[i].index = i;
```

```
    ret = rknn_query(ctx, RKNN_QUERY_CURRENT_OUTPUT_ATTR, &(cur_output_attrs[i]),
sizeof(rknn_tensor_attr));
    if (ret != RKNN_SUCC)
    {
        printf("rknn_query fail! ret=%d\n", ret);
        return -1;
    }
    dump_tensor_attr(&cur_output_attrs[i]);
}
```

注意事项：

- `rknn_set_input_shape`接口要求输入tensor的shape为4维时，fmt使用`NHWC`，非4维时使用`UNDEFINED`。
- 在`rknn_set_input_shapes`尚未调用前，使用带`RKNN_QUERY_CURRENT`前缀的命令查询的shape信息是无效的。

4.推理

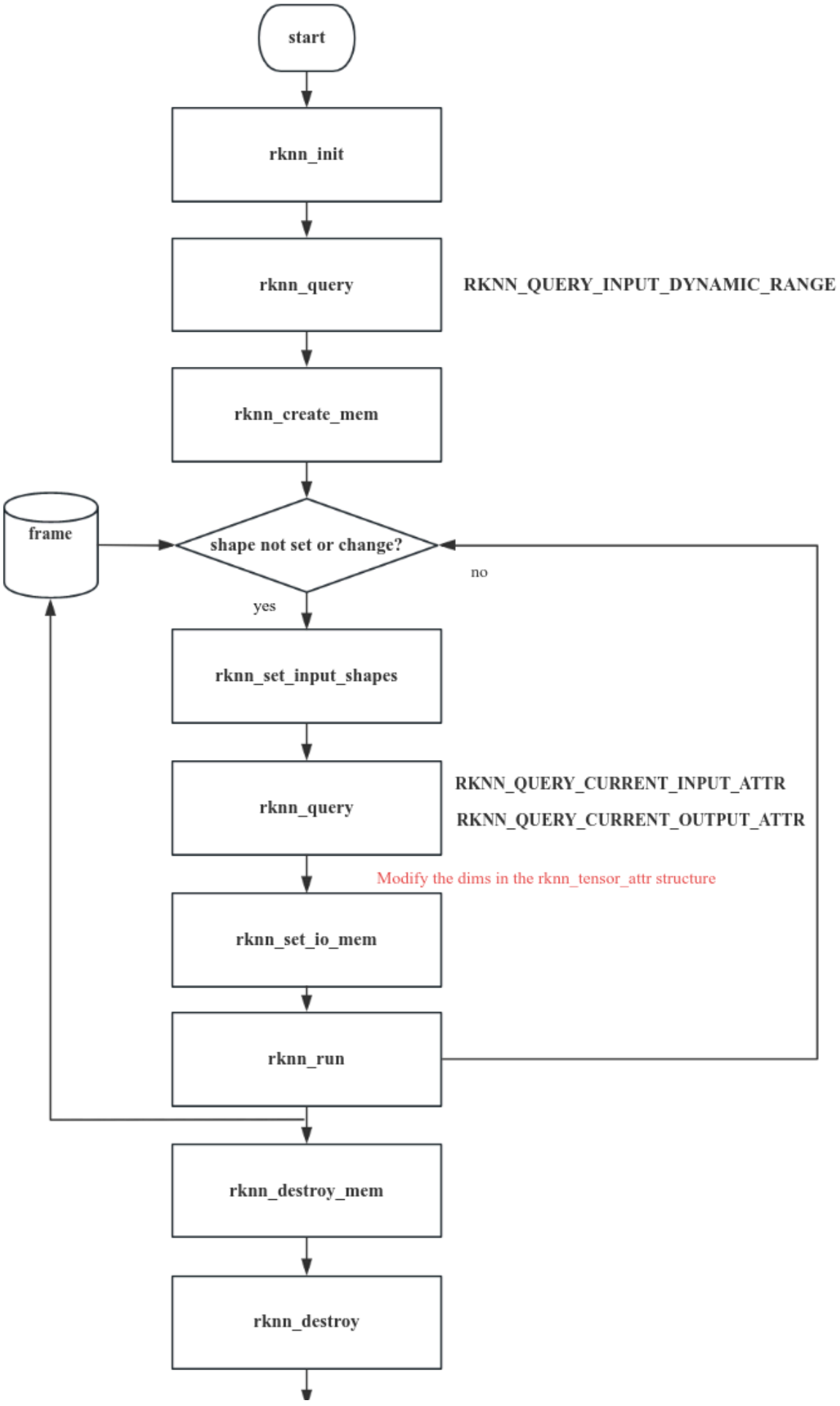
在设置好当前输入shape后，假设输入Tensor的shape信息保存在`cur_input_attrs`数组中，以通用API接口为例，C代码示例如下：

```
// 设置输入信息
rknn_input inputs[io_num.n_input];
memset(inputs, 0, io_num.n_input * sizeof(rknn_input));
for (int i = 0; i < io_num.n_input; i++)
{
    int height = cur_input_attrs[i].fmt == RKNN_TENSOR_NHWC ? cur_input_attrs[i].dims[1] :
cur_input_attrs[i].dims[2];
    int width = cur_input_attrs[i].fmt == RKNN_TENSOR_NHWC ? cur_input_attrs[i].dims[2] :
cur_input_attrs[i].dims[3];
    cv::resize(imgs[i], imgs[i], cv::Size(width, height));
    inputs[i].index = i;
    inputs[i].pass_through = 0;
    inputs[i].type = RKNN_TENSOR_UINT8;
    inputs[i].fmt = RKNN_TENSOR_NHWC;
    inputs[i].buf = imgs[i].data;
    inputs[i].size = imgs[i].total() * imgs[i].channels();
}

// 将输入数据转换成正确的格式后，放到输入缓冲区
ret = rknn_inputs_set(ctx, io_num.n_input, inputs);
if (ret < 0)
{
    printf("rknn_input_set fail! ret=%d\n", ret);
    return -1;
}

// 进行推理
printf("Begin perf ...\n");
double total_time = 0;
for (int i = 0; i < loop_count; ++i)
{
    int64_t start_us = getCurrentTimeUs();
```

```
    ret = rknn_run(ctx, NULL);
    int64_t elapse_us = getCurrentTimeUs() - start_us;
    if (ret < 0)
    {
        printf("rknn run error %d\n", ret);
        return -1;
    }
    total_time += elapse_us / 1000.f;
    printf("%4d: Elapse Time = %.2fms, FPS = %.2f\n", i, elapse_us / 1000.f, 1000.f * 1000.f
/ elapse_us);
}
printf("Avg FPS = %.3f\n", loop_count * 1000.f / total_time);

// 获取输出结果
rknn_output outputs[io_num.n_output];
memset(outputs, 0, io_num.n_output * sizeof(rknn_output));
for (uint32_t i = 0; i < io_num.n_output; ++i)
{
    outputs[i].want_float = 1;
    outputs[i].index = i;
    outputs[i].is_prealloc = 0;
}

ret = rknn_outputs_get(ctx, io_num.n_output, outputs, NULL);
if (ret < 0)
{
    printf("rknn_outputs_get fail! ret=%d\n", ret);
    return ret;
}

//释放输出缓冲区buffer
ret = rknn_outputs_release(ctx, io_num.n_output, outputs);
```

5.4.4.2 零拷贝API

对于零拷贝API而言，初始化成功后，通过`rknn_query()`可以查询RKNN模型支持的输入shape列表，调用`rknn_create_mem()`接口分配的输入和输出内存。接着，通过调用`rknn_set_input_shapes()`接口，传入包含每个输入shape信息的`rknn_tensor_attr`数组指针可以设置当前推理使用的shape。在设置输入shape后，可以再次调用`rknn_query()`查询设置成功后的输入和输出shape。最后，调用`rknn_set_io_mem()`接口设置需要的输入输出内存。每次切换输入shape时，需要再设置一次新的shape，准备新shape大小的数据并再次调用`rknn_set_io_mem()`接口，如果推理前不需要切换输入shape，无需重复调用`rknn_set_input_shapes()`接口。典型用法流程如下图所示：


start
rknn_init
rknn_query
RKNN_QUERY_INPUT_DYNAMIC_RANGE
rknn_create_mem
frame
shape not set or change?
no
yes
rknn_set_input_shapes
rknn_query
RKNN_QUERY_CURRENT_INPUT_ATTR
RKNN_QUERY_CURRENT_OUTPUT_ATTR
Modify the dims in the rknn_tensor_attr structure
rknn_set_io_mem
rknn_run
rknn_destroy_mem
rknn_destroy

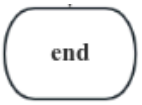


图5-11 动态shape输入接口的零拷贝API调用流程

初始化、查询RKNN模型支持的输入shape组合、设置输入shape使用与上述通用API相同，此处不做赘述。不同之处在于，在设置输入shape后，使用的接口不同。零拷贝推理C代码示例如下：

```
// 创建最大的输入tensor内存
rknn_tensor_mem *input_mems[io_num.n_input];
for (int i = 0; i < io_num.n_input; i++) {
    // default input type is int8 (normalize and quantize need compute in outside)
    // if set uint8, will fuse normalize and quantize to npu
    input_attrs[i].type = RKNN_TENSOR_UINT8;
    // default fmt is NHWC, npu only support NHWC in zero copy mode
    input_attrs[i].fmt = RKNN_TENSOR_NHWC;
    input_mems[i] = rknn_create_mem(ctx, input_attrs[i].size_with_stride);
...
// 创建最大的输出tensor内存
rknn_tensor_mem *output_mems[io_num.n_output];
for (uint32_t i = 0; i < io_num.n_output; ++i) {
    // default output type is depend on model, this require float32 to compute top5
    // allocate float32 output tensor
    int output_size = output_attrs[i].size * sizeof(float);
    output_mems[i] = rknn_create_mem(ctx, output_size);
...
// 加载输入并设置模型输入shape，每次切换输入shape要调用一次
for (int s = 0; s < shape_num; ++s) {
    for (int i = 0; i < io_num.n_input; i++) {
        for (int j = 0; j < input_attrs[i].n_dims; ++j) {
            input_attrs[i].dims[j] = shape_range[i].dyn_range[s][j];
    .....
    ret = rknn_set_input_shapes(ctx, io_num.n_input, input_attrs);
    if (ret < 0) {
        fprintf(stderr, "rknn_set_input_shape error! ret=%d\n", ret);
        return -1;
    ....
    // 获取当前次推理的输入和输出shape
    printf("current input tensors:\n");
    rknn_tensor_attr cur_input_attrs[io_num.n_input];
    memset(cur_input_attrs, 0, io_num.n_input * sizeof(rknn_tensor_attr));
    for (uint32_t i = 0; i < io_num.n_input; i++) {
        cur_input_attrs[i].index = i;
        // query info
        ret = rknn_query(ctx, RKNN_QUERY_CURRENT_INPUT_ATTR, &(cur_input_attrs[i]),
sizeof(rknn_tensor_attr));
        if (ret < 0) {
            printf("rknn_init error! ret=%d\n", ret);
            return -1;
```

```
        dump_tensor_attr(&cur_input_attrs[i]);
        ...
        printf("current output tensors:\n");
    rknn_tensor_attr cur_output_attrs[io_num.n_output];
    memset(cur_output_attrs, 0, io_num.n_output * sizeof(rknn_tensor_attr));
    for (uint32_t i = 0; i < io_num.n_output; i++) {
        cur_output_attrs[i].index = i;
        // query info
        ret = rknn_query(ctx, RKNN_QUERY_CURRENT_OUTPUT_ATTR, &(cur_output_attrs[i]),
sizeof(rknn_tensor_attr));
        if (ret != RKNN_SUCC) {
            printf("rknn_query fail! ret=%d\n", ret);
            return -1;
        ...
        dump_tensor_attr(&cur_output_attrs[i]);
    // 指定NPU核心数量，仅3588/3576支持
    rknn_set_core_mask(ctx, (rknn_core_mask)core_mask);
    // 设置输入信息
    rknn_input inputs[io_num.n_input];
    memset(inputs, 0, io_num.n_input * sizeof(rknn_input));
    std::vector<cv::Mat> resize_imgs;
    resize_imgs.resize(io_num.n_input);
    for (int i = 0; i < io_num.n_input; i++) {
        int height = cur_input_attrs[i].fmt == RKNN_TENSOR_NHWC ? cur_input_attrs[i].dims[1]
: cur_input_attrs[i].dims[2];
        int width = cur_input_attrs[i].fmt == RKNN_TENSOR_NHWC ? cur_input_attrs[i].dims[2]
: cur_input_attrs[i].dims[3];
        int stride = cur_input_attrs[i].w_stride;
        cv::resize(imgs[i], resize_imgs[i], cv::Size(width, height));
        int input_size = resize_imgs[i].total() * resize_imgs[i].channels();
        // 拷贝外部数据到零拷贝输入缓冲区
        if (width == stride)
            memcpy(input_mems[i]->virt_addr, resize_imgs[i].data, input_size);
        else {
            int height = cur_input_attrs[i].dims[1];
            int channel = cur_input_attrs[i].dims[3];
            // copy from src to dst with stride
            uint8_t *src_ptr = resize_imgs[i].data;
            uint8_t *dst_ptr = (uint8_t *)input_mems[i]->virt_addr;
            // width-channel elements
            int src_wc_elems = width * channel;
            int dst_wc_elems = stride * channel;
            for (int b = 0; b < cur_input_attrs[i].dims[0]; b++) {
                for (int h = 0; h < height; ++h) {
                    memcpy(dst_ptr, src_ptr, src_wc_elems);
                    src_ptr += src_wc_elems;
                    dst_ptr += dst_wc_elems;
            ....
    // 更新输入零拷贝缓冲区内存
    for (int i = 0; i < io_num.n_input; i++) {
        cur_input_attrs[i].type = RKNN_TENSOR_UINT8;
        ret = rknn_set_io_mem(ctx, input_mems[i], &cur_input_attrs[i]);
        if (ret < 0) {
```

```
            printf("rknn_set_io_mem fail! ret=%d\n", ret);
            return -1;
     ....
    // 更新输出零拷贝缓冲区内存
    for (uint32_t i = 0; i < io_num.n_output; ++i) {
        // default output type is depend on model, this require float32 to compute top5
        cur_output_attrs[i].type = RKNN_TENSOR_FLOAT32;
        cur_output_attrs[i].fmt = RKNN_TENSOR_NCHW;
        // set output memory and attribute
        ret = rknn_set_io_mem(ctx, output_mems[i], &cur_output_attrs[i]);
        if (ret < 0) {
            printf("rknn_set_io_mem fail! ret=%d\n", ret);
            return -1;
      ....
    // 推理
    printf("Begin perf ...\n");
    double total_time = 0;
    for (int i = 0; i < loop_count; ++i) {
        int64_t start_us = getCurrentTimeUs();
        ret = rknn_run(ctx, NULL);
        int64_t elapse_us = getCurrentTimeUs() - start_us;
        if (ret < 0) {
            printf("rknn run error %d\n", ret);
            return -1;
        ...
        total_time += elapse_us / 1000.f;
        printf("%4d: Elapse Time = %.2fms, FPS = %.2f\n", i, elapse_us / 1000.f, 1000.f *
1000.f / elapse_us);
    printf("Avg FPS = %.3f\n", loop_count * 1000.f / total_time);
```

注意事项：

1. `rknn_set_io_mem()`接口在动态shape情况下，输入buffer的shape和大小说明：

- 初始化完成后和调用`rknn_set_input_shapes()`接口前，`rknn_query()`接口使用`RKNN_QUERY_INPUT_ATTR`和`RKNN_QUERY_OUTPUT_ATTR`查询输入和输出Tensor的shape通常是最大的，用户可以使用这两个命令获取的大小来分配输入和输出内存。若遇到多输入模型，部分输入的shape可能不是最大的，此时需要搜索支持的shape中最大的规格，并分配最大的输入和输出内存。
- 如果输入是非4维度，使用`fmt=UNDEFINED`，传递原始模型输入shape的buffer，大小则根据输入shape和type计算得到。
- 如果输入是4维度，支持使用`fmt=NHWC`或者`NC1HWC2`， 传递`NHWC`或者`NC1HWC2` shape和对应size的buffer(通过`rknn_query`查询相应字段获取shape和size)。
- `rknn_query()`接口中，标志位为`RKNN_QUERY_CURRENT_INPUT_ATTR`和`RKNN_QUERY_CURRENT_OUTPUT_ATTR`时获取原始模型输入/输出的shape，其格式为`NHWC`或者`UNDEFINED`；标志位为`RKNN_QUERY_CURRENT_NATIVE_INPUT_ATTR`和`RKNN_QUERY_CURRENT_NATIVE_OUTPUT_ATTR`时获取NPU以最优性能读取数据时模型输入/输出的shape，其格式为`NHWC`或者`NC1HWC2`。

2. `rknn_set_io_mem()`接口中使用的buffer排列格式为`NHWC`时，`rknn_tensor_attr`中的shape和fmt需按照`RKNN_QUERY_CURRENT_INPUT_ATTR`查询到的信息进行设置；如果使用的buffer排列格式为`NC1HWC2`时，需要按照`RKNN_QUERY_CURRENT_NATIVE_INPUT_ATTR`查询到的信息进行设置。

完整的动态shape C API Demo请参考(https://github.com/airockchip/rknn-toolkit2/tree/master/rknpu2/examples/rknn_dynamic_shape_input_demo)

5.5 自定义算子

5.5.1 自定义算子介绍

RKNN SDK提供了一种自定义算子的机制，它允许开发者在RKNN模型的推理阶段定义和执行自定义的算子。通过实现自定义算子，开发者可以扩展模型功能，并且针对特定硬件（CPU或者GPU）进行优化，以充分利用硬件资源并提高推理速度。同时，开发自定义算子需要深刻的理解深度学习计算原理和目标硬件平台的特性，以确保正确性和性能。目前只支持ONNX模型自定义算子。

RKNN自定义算子主要包括两大步骤：

- 使用RKNN-Toolkit2注册自定义算子并导出RKNN模型。
- 编写自定义算子的C代码实现，通过RKNN API加载注册并执行。

整体流程如下图所示：

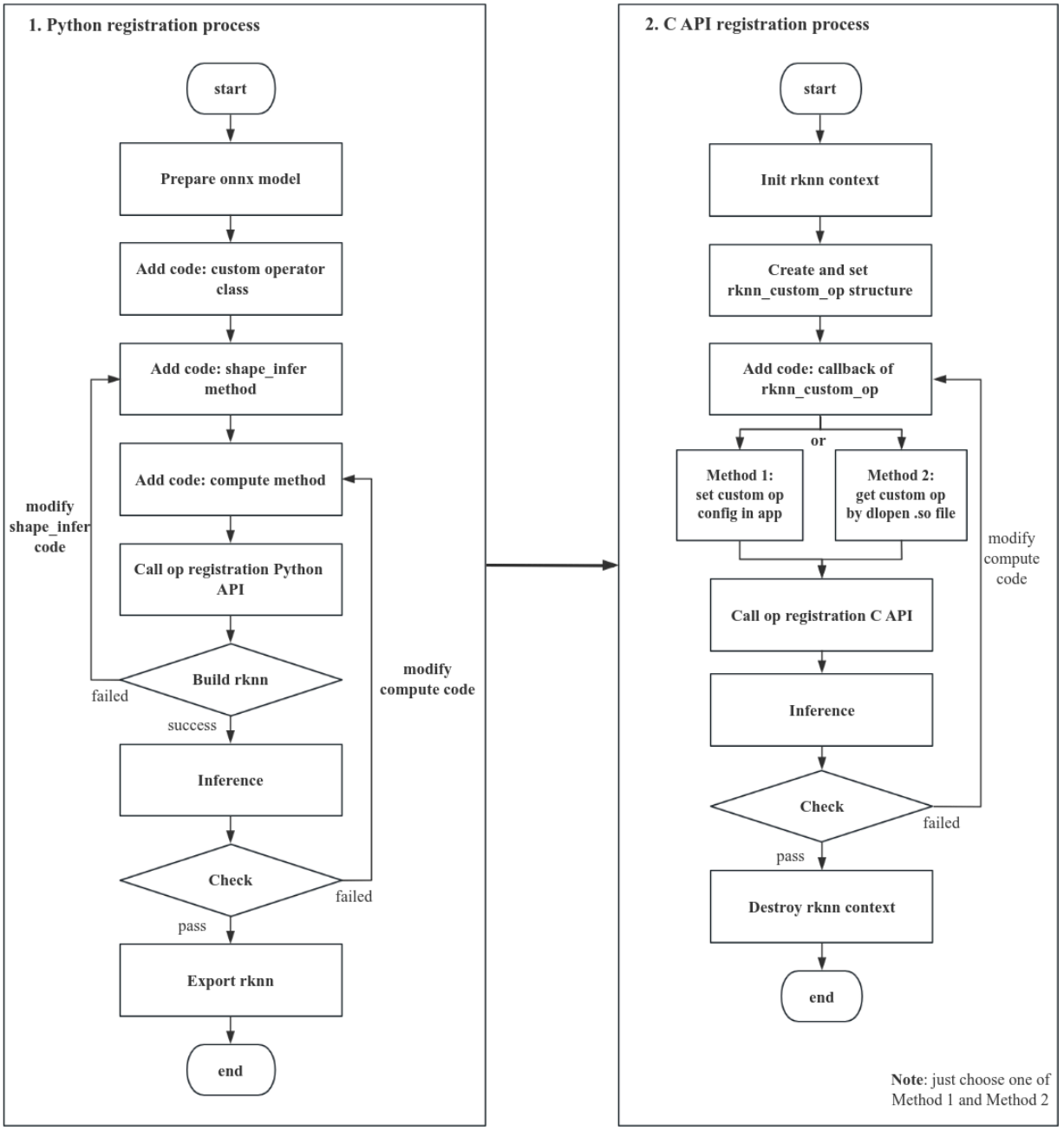


图5-12 注册自定义算子的完整流程

5.5.2 整体流程介绍

5.5.2.1 使用RKNN-Toolkit2注册自定义算子并导出RKNN模型

1. 准备ONNX模型: 按照ONNX模型标准规范, 用户设计自定义算子的op类型,名字, op属性,输入/输出数量,并将该算子插入到ONNX中的拓扑图位置，用户使用ONNX包提供的api设计和导出ONNX模型。
2. 实现自定义算子Python类, 类里主要包括`shape_infer()`和`compute()`两个函数接口, 并调用`rknn.reg_custom_op()`注册该算子。
3. 如果`rknn.build()`执行成功，可以进行推理，否则需要检查步骤2的`shape_infer()`代码实现。然后运行仿真，如果仿真结果正确，可以调用`rknn.export_rknn()`接口导出RKNN模型，否则需要检查步骤2的`compute()`代码。

5.5.2.2 编写自定义算子的C代码实现，通过RKNN API加载注册并执行

1. 根据`rknn_custom_op.h`的`rknn_custom_op`类，编写自定义算子的C代码实现，编写完成后，填写`rknn_custom_op`类的信息。
2. 调用`rknn_register_custom_ops()`注册`rknn_custom_op`类的信息。
3. 参考通用API或零拷贝API的流程，正常构建、推理模型，可以开启模型详细日志和Dump功能确认自定义算子实现的正确性。

5.5.2.3 使用RKNN-Toolkit2连板推理或精度分析

若用户需要对包含自定义算子的模型做Python连板精度分析，需要将自定义算子的回调函数实现代码编译成so后，放在指定的路径，并重启RKNN Server。具体参考5.5.4.4章节。

5.5.3 Python端处理

目前只有ONNX模型支持自定义算子，支持用户**添加非ONNX标准的算子**。

添加非ONNX标准的自定义算子用于新增一个不存在于ONNX算子列表内的新算子，该算子除了要满足ONNX spec规范以外，还要满足以下规则：

- 算子的`op_type`不能与ONNX标准算子相同，推荐以"`cst`"字符开头。
- 算子与其他算子必须要有连接关系，包含各个输入/输出的shape，数据类型等。
- 算子输入属性，支持bool、int32、float32、int64类型的单值或者数组。
- 算子常量输入，支持bool、int32、float32、int64类型，该类型指未量化ONNX模型的数据类型。

因为非ONNX标准算子并不是ONNX SPEC内的标准算子，所以用户需要自行通过ONNX的API或其他框架的API来构建并导出一个包含非ONNX标准算子的ONNX模型。

这边为方便起见，以一个简单的修改Softmax的定义为示例，来构建一个包含非ONNX标准算子`cstSoftmax`的ONNX模型，修改方法如下：

```
import onnx

path="test_softmax.onnx"
model=onnx.load(path)

for node in model.graph.node:
    if node.op_type =="Softmax":
        node.op_type = "cstSoftmax"
        ...   # 修改 cstSoftmax 的属性定义等
onnx.save(model, "./test_softmax_custom.onnx")
```

在构建完包含自定义算子的ONNX模型后，可使用Netron打开该ONNX模型，并检查该自定义算子是否符合自定义算子的规范，满足规范之后，就可以实现该自定义算子的算子类，具体实现如下（以自定义`Softmax`为例）：

```
import numpy as np
from rknn.api.custom_op import get_node_attr
class cstSoftmax:
    op_type = 'cstSoftmax'
    def shape_infer(self, node, in_shapes, in_dtypes):
        out_shapes = in_shapes.copy()
```

```
        out_dtypes = in_dtypes.copy()
        return out_shapes, out_dtypes
    def compute(self, node, inputs):
        x = inputs[0]
        axis = get_node_attr(node, 'axis')
        x_max = np.max(x, axis=axis, keepdims=True)
        tmp = np.exp(x - x_max)
        s = np.sum(tmp, axis=axis, keepdims=True)
        outputs = [tmp / s]
        return outputs
```

- 自定义算子必须为一个Python类。
- 自定义算子类必须包含一个名为`op_type`的字符串变量，与构建的ONNX中自定义算子类型名一致。
- 自定义算子类必须包含成员函数`shape_infer(self, node, in_shapes, in_dtypes)`，函数名、参数名都必须一致，否则报错。该函数用于自定义算子的shape推理，其中，`node`为ONNX的算子节点对象，该对象里包含了自定义算子的属性和输入输出信息；`in_shapes`为该算子所有输入的shape信息，格式为[shape_0, shape_1, ...]，列表内的shape的类型为列表；`in_dtypes`为该算子所有输入的dtype信息，格式为[dtype_0, dtype_1, ...]，列表内的dtype的类型为numpy的dtype类型。另外该函数需要返回该算子所有输出的shape信息和dtype信息，格式与`in_shapes`和`in_dtypes`一致。
- 自定义算子类必须包含成员函数`compute(self, node, inputs)`，函数名和参数名都必须一致，否则报错。该函数用于自定义算子的推理。其中，`node`为ONNX的算子节点对象，该对象里包含了自定义算子的属性和输入输出信息；`inputs`为该算子的输入数据，格式为[array_0, array_1, ...]，列表内的array的类型为numpy的ndarray类型。另外该函数需要返回该算子所有输出的数据，格式与inputs一致。
- 如自定义算子含有自定义的属性，可通过`from rknn.api.custom_op import get_node_attr`来获取自定义算子的属性值。

在编写完自定义算子类后，可以通过`rknn.reg_custom_op()`进行算子类的注册，注册完后，就可以调用`rknn.build()`转换并生成RKNN模型。自定义算子类可以确保模型的转换和推理等功能的正常。具体实现如下（以自定义`Softmax`为例）：

```
from rknn.api import RKNN
# Create RKNN object
rknn = RKNN(verbose=True)

# Pre-process config
print('--> Config model')
rknn.config(mean_values=[103.94, 116.78, 123.68], std_values=[58.82, 58.82, 58.82],
            quant_img_RGB2BGR=True, target_platform='rk3566')
print('done')

print('--> Register cstSoftmax op')
ret = rknn.reg_custom_op(cstSoftmax())
if ret != 0:
    print('Register cstSoftmax op failed!')
    exit(ret)
print('done')

print('--> Loading model')
ret = rknn.load_onnx(model='mobilenet_v2.onnx')
```

```
if ret != 0:
    print('Load model failed!')
    exit(ret)
print('done')

# Build model
print('--> Building model')
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
if ret != 0:
    print('Build model failed!')
    exit(ret)
print('done')
```

`rknn.reg_custom_op()`需要在`rknn.config()`和`rknn.load_xxx()`之间调用。

5.5.4 C API部署

在得到带自定义算子的RKNN模型后，开始调用C API部署。首先，自定义算子的结构体和接口位于`rknn_custom_op.h`头文件，开发者程序需要包含该头文件。注册使用自定义算子的流程如下图所示：

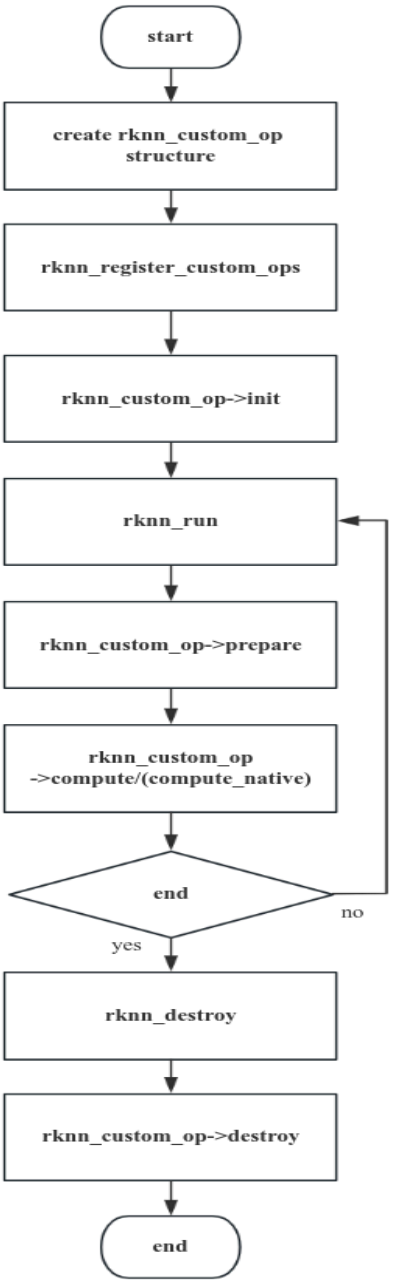


图5-13 注册自定义算子的C API调用流程

5.5.4.1 初始化自定义算子结构体

创建`rknn`上下文后，开发者需要创建`rknn_custom_op`结构体，设置自定义算子信息。算子信息包含以下内容：

- version：算子的版本号。
- target：算子的执行后端，目前支持CPU和GPU。
- op_type: 算子的类型，与ONNX模型中的类型字段相同。
- cl_kernel_name: OpenCL代码的`cl_kernel`函数名。注册GPU算子时必须配置。
- cl_kernel_source：自定义算子的`.cl`文件全路径或者OpenCL kernel字符串。当`cl_source_size=0`, 它表示`.cl`文件全路径；当`cl_source_size>0`，它表示OpenCL kerne代码字符串。注册GPU算子时必须配置。
- cl_source_size: `cl_kernel_source`的大小。大小等于0是特殊情况，它表示`cl_kernel_source`是路径。

- cl_build_options： OpenCL kernel编译选项，以字符串形式传入。注册GPU算子时必须配置。
- init :可选，在`rknn_register_custom_ops`被调用一次。
- prepare：可选，它是预处理回调函数,每次`rknn_run`都会执行`prepare`和`compute/compute_native`回调,执行顺序是`prepare`在前，`compute/compute_native`在后。
- compute：必须实现，算子运算回调函数，它的输入/输出都是`NCHW`的float32格式数据(ONNX模型如果指定输入/输出为int64的数据类型，则int64格式数据)
- compute_native: 保留，请设置成`NULL`。
- destroy:可选，`rknn_destory`中执行一次。
- init/prepare/compute回调函数参数定义规范如下:
  - `rknn_custom_op_context* op_ctx`: op回调函数的上下文信息
  - `rknn_custom_op_tensor* inputs`: op输入tensor数据和信息
  - `uint32_t n_inputs`: op输入个数
  - `rknn_custom_op_tensor* outputs`: op输出tensor数据和信息
  - `uint32_t n_outputs`: op输出个数
- destroy回调函数仅`rknn_custom_op_context* op_ctx`一个参数。

rknn_custom_op_context

包含target（执行后端设备）、GPU上下文、自定义算子私有上下文以及`priv_data`，其中`priv_data`由开发者自行管理（赋值，读写，销毁），GPU上下文包含`cl_context`、`cl_command_queue`、`cl_kernel`指针，可以通过强制类型转换得到对应的OpenCL对象。

`priv_data`是一个用户可选是否配置的指针，通常的用法是用户在`init()`回调函数内创建资源，并将`priv_data`指向该段内存地址，在`prepare()/compute()`回调函数中操作，最终在`destroy()`回调函数内销毁资源。

rknn_custom_op_tensor

表示输入/输出tensor的信息,包含tensor的名称、形状、大小、量化参数、虚拟基地址、fd、数据偏移等信息。

用户在回调`compute()`回调函数内无需创建该算子的输入和输出tensor内存。虚拟地址对应的数据在进入`compute()`回调函数时已经准备好。虚拟地址的计算公式是**Tensor的有效地址=虚拟基地址+数据偏移**，mem成员的virt_addr表示虚拟基地址,mem成员的offset表示数据偏移(以字节为单位)。用户在回调函数内可以读取输入tensor的有效地址，该指向前一层算子已经计算后的输出数据，输出tensor的有效地址指向即将送给下一层算子的输入。

rknn_custom_op_attr

开发者通过调用`rknn_custom_op_get_op_attr()`函数传入属性字段获得属性信息，属性信息用`rknn_custom_op_attr`表示，`rknn_custom_op_attr`中的`void`类型buffer，dtype以及元素数量表示一块内存段，开发者根据dtype使用C/C++将buffer强制转换指针类型可以得到相应数值类型的数组。

5.5.4.1.1 init回调函数

常用于解析算子信息或初始化临时缓冲区或者输入/输出缓冲区buffer。分配临时buffer的`init`回调函数示例代码如下:

- CPU算子

```
/**
 * cpu kernel init callback for custom op
 */
int custom_op_init_callback(rknn_custom_op_context* op_ctx, rknn_custom_op_tensor* inputs,
uint32_t n_inputs,
                            rknn_custom_op_tensor* outputs, uint32_t n_outputs)
{
  printf("custom_op_init_callback\n");
  // create tmp buffer
  float* tmp_buffer = (float*)malloc(inputs[0].attr.n_elems * sizeof(float));
  op_ctx->priv_data = tmp_buffer;
  return 0;
}
```

- GPU算子

```
/**
 * opencl kernel init callback for custom op
 * */
int relu_init_callback_gpu(rknn_custom_op_context* op_ctx, rknn_custom_op_tensor* inputs,
uint32_t n_inputs, rknn_custom_op_tensor* outputs, uint32_t n_outputs)
{
  printf("relu_init_callback_gpu\n");
  // 获取opencl context
  cl_context cl_ctx = (cl_context)op_ctx->gpu_ctx.cl_context;

  // create tmp cl buffer
  cl_mem* memObject = (cl_mem*)malloc(sizeof(cl_mem) * 2);
  memObject[0]      = clCreateBuffer(cl_ctx, CL_MEM_READ_WRITE, inputs[0].attr.size, NULL,
NULL);
  memObject[1]      = clCreateBuffer(cl_ctx, CL_MEM_READ_WRITE, outputs[0].attr.size, NULL,
NULL);
  op_ctx->priv_data = memObject;
  return 0;
}
```

5.5.4.1.2 prepare回调函数

该回调函数每帧推理都会调用，目前为预留实现。

5.5.4.1.3 compute回调函数

它是自定义算子的计算函数,开发者必须完成输入/输出是`NCHW`或`UNDEFINED`格式float32数据类型输入输出的核函数。

1. `compute`回调（CPU）假设开发者想实现一个自定义层，完成softmax功能，CPU算子`compute`函数示例如下：

```
/**
 * float32 kernel implemetation sample for custom op
 * */
int compute_custom_softmax_float32(rknn_custom_op_context* op_ctx, rknn_custom_op_tensor*
inputs, uint32_t n_inputs,
                                   rknn_custom_op_tensor* outputs, uint32_t n_outputs)
```

```
{
  unsigned char*      in_ptr   = (unsigned char*)inputs[0].mem.virt_addr +
inputs[0].mem.offset;
  unsigned char*      out_ptr  = (unsigned char*)outputs[0].mem.virt_addr +
outputs[0].mem.offset;
  int                 axis     = 0;
  const float*        in_data  = (const float*)in_ptr;
  float*              out_data = (float*)out_ptr;
  std::string         name     = "";
  rknn_custom_op_attr op_name;
  rknn_custom_op_get_op_attr(op_ctx, "name", &op_name);
  if (op_name.n_elems > 0 && op_name.dtype == RKNN_TENSOR_UINT8) {
    name = (char*)op_name.data;
  }

  rknn_custom_op_attr op_attr;
  rknn_custom_op_get_op_attr(op_ctx, "axis", &op_attr);
  if (op_attr.n_elems == 1 && op_attr.dtype == RKNN_TENSOR_INT64) {
    axis = ((int64_t*)op_attr.data)[0];
  }

  printf("op name = %s, axis = %d\n", name.c_str(), axis);
  float* tmp_buffer = (float*)op_ctx->priv_data;
  // kernel implemetation for custom op
  {
    int inside  = 1;
    int outside = 1;
    int channel = 1;

    while (axis < 0) {
      axis += inputs[0].attr.n_dims;
    }

    for (int i = 0; i < axis; i++) {
      outside *= inputs[0].attr.dims[i];
    }
    channel = inputs[0].attr.dims[axis];
    for (int i = axis; i < inputs[0].attr.n_dims; i++) {
      inside *= inputs[0].attr.dims[i];
    }

    for (int y = 0; y < outside; y++) {
      const float* src_y    = in_data + y * inside;
      float*       dst_y    = out_data + y * inside;
      float        max_data = -FLT_MAX;
      float        sum_data = 0.0f;

      for (int i = 0; i < inside; ++i) {
        max_data = fmaxf(max_data, src_y[i]);
      }
      for (int i = 0; i < inside; ++i) {
        tmp_buffer[i] = expf(src_y[i] - max_data);
        sum_data += tmp_buffer[i];
```

```
      }
      for (int i = 0; i < inside; ++i) {
        dst_y[i] = tmp_buffer[i] / sum_data;
      }
    }
  }
  return 0;
}
```

2. compute回调函数（GPU）

对于GPU算子，开发者可以在回调函数中完成以下步骤：

- 开发者从`rknn_custom_op_context`里的`gpu_ctx`中获取OpenCL的`cl_context`， `cl_command_queue`以及`cl_kernel`对象，此过程需要开发者做数据类型转换。
- 如有必要，用户自行创建的op输入或输出的`cl_mem`对象缓冲区。
- 设置`cl_kernel`的函数参数。
- OpenCL kernel的函数参数的输入buffer数据目前只能支持float，其他类型暂时还不支持。
- 对于使用零拷贝的情况下，调用`clImportMemoryARM`可以自行协助用户把输入tensor的内存映射到OpenCL的`cl_mem`结构体中，输入tensor已包含输入数据，用户不需要自行再拷贝一次。该过程也可以在init回调函数中处理，然后将`cl_mem`结构体记录到`priv_data`成员，最后在`compute`回调中读取`priv_data`并使用它。
- 以阻塞的形式运行`cl_kernel`。
- CL kernel内的输入数据都是以`NCHW`形式排布给出。
- 如果在GPU运算完后，开发者需要CPU访问数据，需要通过调用`rknn_mem_sync`函数刷新输出Tensor的cache后再读取数据。

假设开发者想实现一个自定义层，完成relu功能，GPU算子compute函数示例如下：

```
/**
 * opencl kernel init callback for custom op
 * */
int compute_custom_relu_float32(rknn_custom_op_context* op_ctx, rknn_custom_op_tensor*
inputs, uint32_t num_inputs,
                                rknn_custom_op_tensor* outputs, uint32_t num_outputs)
{
  std::string         name = "";
  rknn_custom_op_attr op_name;
  rknn_custom_op_get_op_attr(op_ctx, "name", &op_name);
  if (op_name.n_elems > 0 && op_name.dtype == RKNN_TENSOR_UINT8) {
    name = (char*)op_name.data;
  }

  // get context
  cl_context cl_ctx = (cl_context)op_ctx->gpu_ctx.cl_context;

  // get command queue
  cl_command_queue queue = (cl_command_queue)op_ctx->gpu_ctx.cl_command_queue;

  // get kernel
```

```
  cl_kernel kernel = (cl_kernel)op_ctx->gpu_ctx.cl_kernel;

  // import input/output buffer
  const cl_import_properties_arm props[3] = {
    CL_IMPORT_TYPE_ARM,
    CL_IMPORT_TYPE_DMA_BUF_ARM,
    0,
  };

  cl_int status;
  cl_mem inObject = clImportMemoryARM(cl_ctx, CL_MEM_READ_WRITE, props, &inputs[0].mem.fd,
                                      inputs[0].mem.offset + inputs[0].mem.size, &status);
  if (status != CL_SUCCESS) {
    printf("Tensor: %s clImportMemoryARM failed\n", inputs[0].attr.name);
  }
  cl_mem outObject = clImportMemoryARM(cl_ctx, CL_MEM_READ_WRITE, props, &outputs[0].mem.fd,
                                       outputs[0].mem.offset + outputs[0].mem.size,
&status);
  if (status != CL_SUCCESS) {
    printf("Tensor: %s clImportMemoryARM failed\n", outputs[0].attr.name);
  }

  int          in_type_bytes  = get_type_bytes(inputs[0].attr.type);
  int          out_type_bytes = get_type_bytes(outputs[0].attr.type);
  int          in_offset      = inputs[0].mem.offset / in_type_bytes;
  int          out_offset     = outputs[0].mem.offset / out_type_bytes;
  unsigned int elems          = inputs[0].attr.n_elems;

  // set kernel args
  int argIndex = 0;
  clSetKernelArg(kernel, argIndex++, sizeof(cl_mem), &inObject);
  clSetKernelArg(kernel, argIndex++, sizeof(cl_mem), &outObject);
  clSetKernelArg(kernel, argIndex++, sizeof(int), &in_offset);
  clSetKernelArg(kernel, argIndex++, sizeof(int), &out_offset);
  clSetKernelArg(kernel, argIndex++, sizeof(unsigned int), &elems);

  // set global worksize
  const size_t global_work_size[3] = {elems, 1, 1};

  // enqueueNDRangeKernel
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL);

  // finish command queue
  clFinish(queue);

  // //cpu access data after sync to device
  // rknn_mem_sync(&outputs[0].mem, RKNN_MEMORY_SYNC_FROM_DEVICE);
  // // save output npy
  // char output_path[PATH_MAX];
  // sprintf(output_path, "%s/cpu_output%d.npy", ".", 0);
  // unsigned char* out_data = (unsigned
char*)outputs[0].mem.virt_addr+outputs[0].mem.offset;
  // save_npy(output_path, (float*)out_data, &inputs[0].attr);
```

```
  return 0;
}
```

5.5.4.1.4 destroy回调函数

常用于销毁自定义算子的临时缓冲区或输入/输出buffer。销毁临时buffer的示例代码如下:

- CPU算子

```
/**
 * cpu kernel destroy callback for custom op
 */
int custom_op_destroy_callback(rknn_custom_op_context* op_ctx)
{
  printf("custom_op_destroy_callback\n");
  // clear tmp buffer
  free(op_ctx->priv_data);
  return 0;
}
```

- GPU算子

```
/**
 * opencl kernel destroy callback for custom op
 */
int relu_destroy_callback_gpu(rknn_custom_op_context* op_ctx)
{
  // clear tmp buffer
  printf("relu_destroy_callback_gpu\n");
  cl_mem* memObject = (cl_mem*)op_ctx->priv_data;
  clReleaseMemObject(memObject[0]);
  clReleaseMemObject(memObject[1]);
  free(memObject);
  return 0;
}
```

5.5.4.2 注册自定义算子

在设置完`rknn_custom_op`结构体后，需要调用`rknn_register_custom_ops()`将其注册到`rknn_context`中，该接口支持同时注册多个自定义算子。

在完成CPU的`compute`回调函数后，注册一个名为"cstSoftmax"和“ArgMax”的CPU自定义算子的代码示例如下：

- CPU算子

```
// CPU operators
rknn_custom_op user_op[2];
memset(user_op, 0, 2 * sizeof(rknn_custom_op));
strncpy(user_op[0].op_type, "cstSoftmax", RKNN_MAX_NAME_LEN - 1);
user_op[0].version = 1;
user_op[0].target  = RKNN_TARGET_TYPE_CPU;
user_op[0].init    = custom_op_init_callback;
user_op[0].compute = compute_custom_softmax_float32;
```

```
user_op[0].destroy = custom_op_destroy_callback;

strncpy(user_op[1].op_type, "ArgMax", RKNN_MAX_NAME_LEN - 1);
user_op[1].version = 1;
user_op[1].target  = RKNN_TARGET_TYPE_CPU;
user_op[1].init    = custom_op_init_callback;
user_op[1].compute = compute_custom_argmax_float32;
user_op[1].destroy = custom_op_destroy_callback;

ret = rknn_register_custom_ops(ctx, user_op, 2);
if (ret < 0) {
printf("rknn_register_custom_ops fail! ret = %d\n", ret);
return -1;
}
```

- GPU算子

对于GPU算子而言，支持以常量字符串或者文件路径的两种方式注册OpenCL kernel。当`rknn_custom_op`结构体中的`cl_source_size`等于0时，`cl_kernel_source`表示OpenCL kernel的文件路径，当`cl_source_size`大于0时`cl_kernel_source`表示OpenCL kernel函数字符串。以字符串保存的relu功能的OpenCL kernel的示例代码如下:

```
char* cl_kernel_source = "#pragma OPENCL EXTENSION cl_arm_printf : enable"
     "#pragma OPENCL EXTENSION cl_khr_fp16 : enable "
     " __kernel void relu_float(__global const float* input, __global float* output, int "
     "in_offset, int out_offset, const unsigned int elems) "
     "{"
     "  int gid = get_global_id(0);"
     "  if (gid < elems) {"
     "    float in_value           = input[in_offset + gid];"
     "    output[out_offset + gid] = in_value >= 0.f ? in_value : 0.f;"
     "  }"
     "}"
     "__kernel void relu_half(__global const half* input, __global half* output, int
in_offset, "
     "int out_offset, const unsigned int elems)"
     "{"
     "  int gid = get_global_id(0);"
     "  if (gid < elems) {"
     "    half in_value            = input[in_offset + gid];"
     "    output[out_offset + gid] = in_value >= 0.f ? in_value : 0.f;"
     "  }"
     "}";
```

在完成OpenCL kernel函数以及GPU的compute回调函数后，可以设置`rknn_custom_op`结构体数组并注册GPU算子，注册GPU算子示例代码如下:

```
// GPU operators
rknn_custom_op user_op[1];
memset(user_op, 0, sizeof(rknn_custom_op));
strncpy(user_op->op_type, "cstSoftmax", RKNN_MAX_NAME_LEN - 1);
user_op->version = 1;
user_op->target  = RKNN_TARGET_TYPE_GPU;
```

```
user_op->init    = relu_init_callback_gpu;
user_op->compute = compute_custom_relu_float32;
user_op->destroy = relu_destroy_callback_gpu;
#ifdef LOAD_FROM_PATH
user_op->cl_kernel_source = "./custom_op.cl";
user_op->cl_source_size   = 0;
#else
user_op->cl_kernel_source = cl_kernel_source;
user_op->cl_source_size   = strlen(cl_kernel_source);
#endif
strcpy(user_op->cl_kernel_name, "relu_float");

ret = rknn_register_custom_ops(ctx, user_op, 1);
if (ret < 0) {
printf("rknn_register_custom_ops fail! ret = %d\n", ret);
return -1;
}
```

注册调用该接口前要明确自定义算子的`op_type`，准备好算子信息并配置`rknn_custom_op`数组。每个类型的自定义算子要调用一次注册接口，网络中同一类型的算子仅调用一次。

5.5.4.3 模型推理

在注册完所有算子后，可以使用通用API或零拷贝API流程完成推理。

5.5.4.4 连板精度分析

自定义算子的连板调试功能要求rknn_server版本>=1.6.0。

连板调试时，RKNN Server会采用dlopen的方式从特定目录打开用户编译好的自定义算子插件库来获取算子信息，对于插件库方式注册自定义算子，要求用户必须实现一个名为`get_rknn_custom_op`的函数。

若用户需要对包含自定义算子的模型做连板精度分析，具体步骤如下：

1. 实现一个`get_rknn_custom_op()`函数和必须的回调函数，并编译成对应系统的库，编译的插件库名称必须以"librkcst_"为前缀，例如库名是`librkcst_relu.so`。
2. 插件放到`/vendor/lib64/`（Android arm64-v8a）或`/usr/lib/rknpu/op_plugins`（Linux）
3. 主机端或者上位机使用RKNN-Toolkit2的Python接口执行连板精度分析。

`get_rknn_custom_op`函数的示例代码如下：

```
std::vector<std::string> get_all_plugin_paths(std::string plugin_dir)
{
  std::vector<std::string> plugin_paths;
  if (access(plugin_dir.c_str(), 0) != 0) {
    fprintf(stderr, "Can not access plugin directory: %s, please check it!\n",
plugin_dir.c_str());
  }

  DIR*           dir;
  struct dirent* ent;
  const char*    prefix = RKNN_CSTOP_PLUGIN_PREFIX; // 所有库文件名应该以此前缀开头
```

```
  if ((dir = opendir(plugin_dir.c_str())) != NULL) {
    while ((ent = readdir(dir)) != NULL) {
      if (ent->d_type == DT_REG) {
        const char* filename = ent->d_name;
        size_t      len      = strlen(filename);

        if (len > 10 && strncmp(filename, prefix, strlen(prefix)) == 0) {
          printf("Found plugin: %s file in %s\n", filename, plugin_dir.c_str());
          plugin_paths.push_back(plugin_dir + "/" + filename);
        }
      }
    }
    closedir(dir);
  } else {
    fprintf(stderr, "Unable to open directory");
  }

  return plugin_paths;
}

// the default path of the custom operator plugin libraries
std::string plugin_dir =
#if defined(__ANDROID__)
#  if defined(__aarch64__)
    "/vendor/lib64/";
#  else
    "/vendor/lib/";
#  endif // __aarch64__
#elif defined(__linux__)
    "/usr/lib/rknpu/op_plugins/";
#endif

std::vector<std::string> plugin_paths = get_all_plugin_paths(plugin_dir);
std::vector<void*> so_handles;
for (auto path : plugin_paths) {
printf("load plugin %s\n", path.c_str());
void* plugin_lib = dlopen(path.c_str(), RTLD_NOW);
char* error      = dlerror();
if (error != NULL) {
    fprintf(stderr, "dlopen %s fail: %s.\nPlease try to set 'export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:%s'\n",
            path.c_str(), error, plugin_dir.c_str());
    dlclose(plugin_lib);
    return -1;
}
printf("dlopen %s successfully!\n", path.c_str());
get_custom_op_func custom_op_func = (get_custom_op_func)dlsym(plugin_lib,
"get_rknn_custom_op");
error                             = dlerror();
if (error != NULL) {
    fprintf(stderr, "dlsym fail: %s\n", error);
    dlclose(plugin_lib);
    return -1;
```

```
}

rknn_custom_op* user_op = custom_op_func();
ret                     = rknn_register_custom_ops(ctx, user_op, 1);
if (ret < 0) {
    printf("rknn_register_custom_ops fail! ret = %d\n", ret);
    return -1;
}
so_handles.push_back(plugin_lib);
}
```

插件库有如下注意事项：

- 一个自定义算子插件库只能注册一个自定义算子，如果需要注册多个自定义算子，需要创建多个插件库。
- 插件库的名称必须以"librkcst_"开头，以.so结尾。
- 如果是C++代码实现的插件库，C接口中为了正确打开函数符号，要在函数定义前后加入如下宏：

```
#ifdef __cplusplus
extern "C" {
#endif

//code

#ifdef __cplusplus
} // extern "C"
#endif
```

- 如果dlopen插件库失败,需要检查是否设置了`LD_LIBRARY_PATH`环境变量，并且检查插件库所在目录是否在环境变量指定的路径。

5.6 多Batch使用说明

5.6.1 多Batch原理

RK3588 NPU内部有3个核心，RK3576 NPU内部有2个核心，为了更高效得利用多核性能，提供了多batch推理功能。当开启多batch推理时，内部会调用`rknn_dup_context`将`context`进行拷贝（`rknn_dup_context`只会对`context`的Internal 进行拷贝，Weight会复用）。当`rknn_batch_size=2`时，会拷贝1份，当`rknn_batch_size >=3`时，会拷贝2份（同一时刻最多只有3个核心工作，为了避免内存浪费只拷贝2份）。**每个context core_mask 会设置成0，让多核内部自动调度**。当执行`rknn_run()`时，内部会起一个线程池，同一时刻调用3个线程同时对3个`context`进行推理。

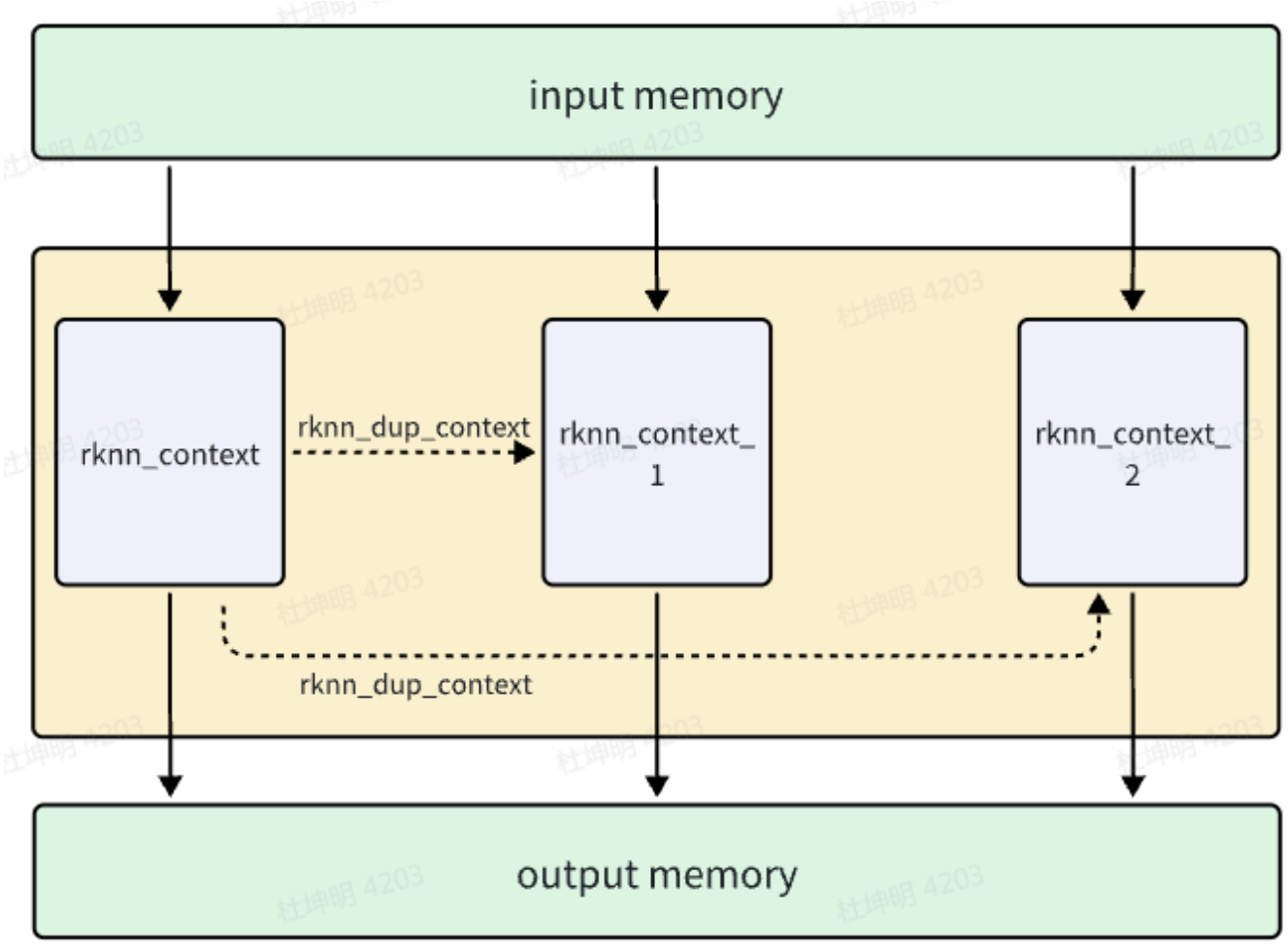


图5-14 多batch内部原理图

5.6.2 多Batch使用方式

多batch使用方式如下：

1. Python 端开启多 batch 设置：

```
ret = rknn.build(do_quantization=True, dataset='./dataset.txt', rknn_batch_size=3)
```

为了达到最优性能，RK3588建议`rknn_batch_size`为3的倍数，RK3576建议`rknn_batch_size`为2的倍数。

2. 建议使用**零拷贝接口**

5.6.3 多Batch输入输出设置

当开启多batch功能时，查询出来的输入输出size是未开启时的`rknn_batch_size`倍。内部每个`context`会各自算自己的一个输入偏移量，按照这个输入偏移量取输入数据做推理，然后各自算自己的一个输出偏移量，按照这个输出偏移量写到各自的输出。以第二个batch为例，输入偏移量是查询出来的`input_size`除以`rknn_batch_size`，输出偏移量是查询出来的`output_size`除以`rknn_batch_size`。

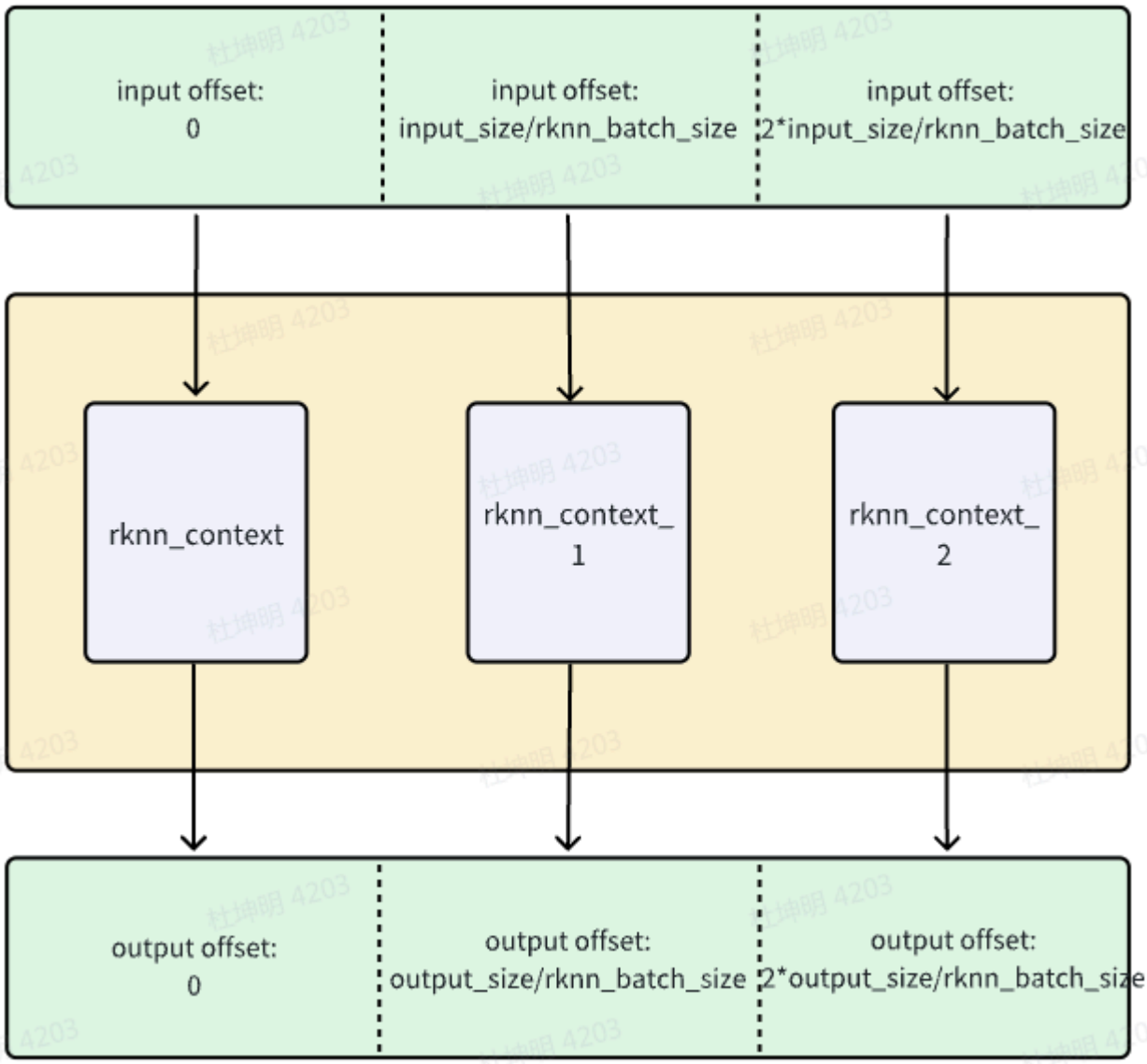


图5-15 多batch内部输入输出地址偏移图

5.7 RK3588/RK3576 NPU SRAM使用说明

- RK3588 SOC内部含有1MB的SRAM，其中有956KB可供给SOC上各个IP所使用。
- RK3576 SOC内部含有512KB的SRAM，其中有480KB可供给SOC上各个IP所使用。
- SRAM可以帮助RKNPU应用减轻DDR带宽压力，但对推理耗时可能有一定影响。

5.7.1 板端环境要求

5.7.1.1 内核环境要求

- RKNPU驱动版本>=0.9.2
- 内核config需要开启`CONFIG_ROCKCHIP_RKNPU_SRAM=y`
  - Android系统config路径如下：

```
<path-to-your-kernel>/arch/arm64/configs/rockchip_defconfig
```

  - Linux系统config路径如下：

```
    <path-to-your-kernel>/arch/arm64/configs/rockchip_linux_defconfig
```

- 内核相应DTS需要从系统SRAM中分配给RKNPU使用
  - 从系统分配需求大小的SRAM给RKNPU，RK3588最大可分配956KB，RK3576最大可分配480KB，且大小需要4K对齐。
  - 注意：默认系统中可能已为其他IP分配SRAM，比如编解码模块，各IP分配的SRAM区域不能重叠，否则会存在同时读写出现数据错乱现象。
  - 如下为RK3588分配956KB和RK3576分配480KB给RKNPU的例子

RK3588示例：

```
syssram: sram@ff001000 {
    compatible = "mmio-sram";
    reg = <0x0 0xff001000 0x0 0xef000>;

    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0x0 0xff001000 0xef000>;
    /* 分配RKNPU SRAM /    / start address and size should be 4k algin */
    rknpu_sram: rknpu_sram@0 {
        reg = <0x0 0xef000>; // 956KB
    };
};
```

- 把分配的SRAM挂到RKNPU节点，修改如下所示的dtsi文件：

```
<path-to-your-kernel>/arch/arm64/boot/dts/rockchip/rk3588s.dtsi
```

```
rknpu: npu@fdab0000 {
    compatible = "rockchip,rk3588-rknpu";
    /* ... /    / 增加RKNPU sram的引用 */
    rockchip,sram = <&rknpu_sram>;
    status = "disabled";
};
```

RK3576 示例：

```
sram: sram@3ff88000 {
    compatible = "mmio-sram";
    reg = <0x0 0x3ff88000 0x0 0x78000>;

    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0x0 0x3ff88000 0x78000>;
    /* 分配RKNPU SRAM /    / start address and size should be 4k algin */
    rknpu_sram: rknpu-sram@0 {
            reg = <0x0 0x78000>; // 480KB
    };
};
```

- 把分配的SRAM挂到RKNPU节点，修改如下所示的dtsi文件：

```
<path-to-your-kernel>/arch/arm64/boot/dts/rockchip/rk3576.dtsi
```

```
rknpu: npu@27700000 {
    compatible = "rockchip,rk3576-rknpu";
    /* ... /    / 增加RKNPU sram的引用 */
    rockchip,sram = <&rknpu_sram>;
    status = "disabled";
};
```

5.7.1.2 RKNN SDK版本要求

- RKNPU Runtime库(librknnrt.so)版本>=1.6.0

5.7.2 使用方法

在`rknn_init()`接口的flags参数指定`RKNN_FLAG_ENABLE_SRAM`即可在该`context`中开启SRAM。

例如：

```
ret = rknn_init(&ctx, rknn_model, size, RKNN_FLAG_ENABLE_SRAM, NULL);
```

当设置`RKNN_FLAG_ENABLE_SRAM`时，将从系统可用的SRAM中分配尽可能多的内存做为模型的Internal Tensor内存。

注意：

- 当SRAM被某一`rknn_context`占用后，其他的`rknn_context`不支持复用该段的SRAM。`rknn_api.h`中的`RKNN_FLAG_SHARE_SRAM`功能暂未实现。
- 当某个`rknn_context`未占用全部的SRAM时，剩余的SRAM可以给其他的`rknn_context`使用。

5.7.3 调试方法

5.7.3.1 SRAM是否启用查询

通过开机内核日志查看SRAM是否启用，包含为RKNPU指定SRAM的地址范围和大小信息，如下所示：

```
rk3588_s:/ # dmesg | grep rknpu -i
RKNPU fdab0000.npu: RKNPU: sram region: [0x00000000ff001000, 0x00000000ff0f0000), sram size:
0xef000
```

5.7.3.2 SRAM使用情况查询

- 可通过节点查询SRAM的使用情况
- 如下为未使用SRAM的位图表，每个点表示4K大小

```
rk3588_s:/ # cat /sys/kernel/debug/rknpu/mm
SRAM bitmap: "*" - used, "." - free (1bit = 4KB)
[000] [..............................]
[001] [..............................]
[002] [..............................]
[003] [..............................]
[004] [..............................]
[005] [..............................]
[006] [..............................]
[007] [..............]
SRAM total size: 978944, used: 0, free: 978944
# 单位为Byte
```

- 如下为分配使用512KB后的SRAM位图表

```
rk3588_s:/ # cat /sys/kernel/debug/rknpu/mm
SRAM bitmap: "*" - used, "." - free (1bit = 4KB)
[000] [******************************]
[001] [******************************]
[002] [******************************]
[003] [******************************]
[004] [..............................]
[005] [..............................]
[006] [..............................]
[007] [..............]
SRAM total size: 978944, used: 524288, free: 454656
# 单位为Byte
```

5.7.3.3 通过RKNN API查询SRAM大小

- 通过`rknn_query()`的`RKNN_QUERY_MEM_SIZE`接口查询SRAM大小信息

```
typedef struct _rknn_mem_size {
    uint32_t total_weight_size;
    uint32_t total_internal_size;
    uint64_t total_dma_allocated_size;
    uint32_t total_sram_size;
    uint32_t free_sram_size;
    uint32_t reserved[10];
} rknn_mem_size;
```

- 其中，`total_sram_size`表示：系统给RKNPU分配的SRAM总大小，单位是Byte。
- `free_sram_size`表示：剩余RKNPU能使用的SRAM大小，单位是Byte。

5.7.3.4 查看模型SRAM的占用情况

- 板端环境中，RKNN应用运行前设置如下环境变量，可打印SRAM使用预测情况：

```
export RKNN_LOG_LEVEL=3
```

- Internal分配SRAM的逐层占用情况，如下日志所示：

```
--------------------------------------------------------------------------------
Total allocated Internal SRAM Size: 524288, Addr: [0xff3e0000, 0xff460000)
--------------------------------------------------------------------------------
--------------------------------------------------------------------------+----------------------
------------+-----------
ID  User           Tensor   DataType OrigShape      NativeShape      |     [Start       End)
      Size |    SramHit
--------------------------------------------------------------------------+----------------------
------------+-----------
1   ConvRelu       input0   INT8     (1,3,224,224)  (1,1,224,224,3)  | 0xff3b0000 0xff3d4c00
0x00024c00 | \
2   ConvRelu       output2  INT8     (1,32,112,112) (1,2,112,112,16) | 0xff404c00 0xff466c00
0x00062000 | 0x0005b400
3   ConvRelu       output4  INT8     (1,32,112,112) (1,4,112,112,16) | 0xff466c00 0xff52ac00
0x000c4000 | 0x00000000
4   ConvRelu       output6  INT8     (1,64,112,112) (1,4,112,112,16) | 0xff52ac00*0xff5eec00
0x000c4000 | 0x00000000
5   ConvRelu       output8  INT8     (1,64,56,56)   (1,4,56,56,16)   | 0xff3e0000 0xff411000
0x00031000 | 0x00031000
6   ConvRelu       output10 INT8     (1,128,56,56)  (1,8,56,56,16)   | 0xff411000 0xff473000
0x00062000 | 0x0004f000
7   ConvRelu       output12 INT8     (1,128,56,56)  (1,8,56,56,16)   | 0xff473000 0xff4d5000
0x00062000 | 0x00000000
8   ConvRelu       output14 INT8     (1,128,56,56)  (1,8,56,56,16)   | 0xff3e0000 0xff442000
0x00062000 | 0x00062000
9   ConvRelu       output16 INT8     (1,128,28,28)  (1,8,28,28,16)   | 0xff442000 0xff45a800
0x00018800 | 0x00018800
10  ConvRelu       output18 INT8     (1,256,28,28)  (1,16,28,28,16)  | 0xff3e0000 0xff411000
0x00031000 | 0x00031000
11  ConvRelu       output20 INT8     (1,256,28,28)  (1,16,28,28,16)  | 0xff411000 0xff442000
0x00031000 | 0x00031000
12  ConvRelu       output22 INT8     (1,256,28,28)  (1,16,28,28,16)  | 0xff3e0000 0xff411000
0x00031000 | 0x00031000
13  ConvRelu       output24 INT8     (1,256,14,14)  (1,16,14,14,16)  | 0xff411000 0xff41d400
0x0000c400 | 0x0000c400
14  ConvRelu       output26 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3e0000 0xff3f8800
0x00018800 | 0x00018800
15  ConvRelu       output28 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3f8800 0xff411000
0x00018800 | 0x00018800
16  ConvRelu       output30 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3e0000 0xff3f8800
0x00018800 | 0x00018800
17  ConvRelu       output32 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3f8800 0xff411000
0x00018800 | 0x00018800
18  ConvRelu       output34 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3e0000 0xff3f8800
0x00018800 | 0x00018800
19  ConvRelu       output36 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3f8800 0xff411000
0x00018800 | 0x00018800
20  ConvRelu       output38 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3e0000 0xff3f8800
0x00018800 | 0x00018800
21  ConvRelu       output40 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3f8800 0xff411000
0x00018800 | 0x00018800
22  ConvRelu       output42 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3e0000 0xff3f8800
0x00018800 | 0x00018800
```

```
23  ConvRelu       output44 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3f8800 0xff411000
0x00018800 | 0x00018800
24  ConvRelu       output46 INT8     (1,512,14,14)  (1,32,14,14,16)  | 0xff3e0000 0xff3f8800
0x00018800 | 0x00018800
25  ConvRelu       output48 INT8     (1,512,7,7)    (1,33,7,7,16)    | 0xff3f8800 0xff3ff000
0x00006800 | 0x00006800
26  ConvRelu       output50 INT8     (1,1024,7,7)   (1,67,7,7,16)    | 0xff3e0000 0xff3ed000
0x0000d000 | 0x0000d000
27  ConvRelu       output52 INT8     (1,1024,7,7)   (1,67,7,7,16)    | 0xff3ed000 0xff3fa000
0x0000d000 | 0x0000d000
28  AveragePool    output54 INT8     (1,1024,7,7)   (1,67,7,7,16)    | 0xff3e0000 0xff3ed000
0x0000d000 | 0x0000d000
29  Conv           output55 INT8     (1,1024,1,1)   (1,64,1,1,16)    | 0xff3ed000 0xff3ed400
0x00000400 | 0x00000400
30  Softmax        output56 INT8     (1,1000,1,1)   (1,64,1,1,16)    | 0xff3e0000 0xff3e0400
0x00000400 | 0x00000400
31  OutputOperator output57 FLOAT    (1,1000,1,1)   (1,1000,1,1)     | 0xff3ae000 0xff3aefa0
0x00000fa0 | \
-------------------------------------------------------------------------+---------------------
------------+-----------
---------------------------------------
Total Weight Memory Size: 4260864
Total Internal Memory Size: 2157568
Predict Internal Memory RW Amount: 11068320
Predict Weight Memory RW Amount: 4260832
Predict SRAM Hit RW Amount: 6688768
---------------------------------------
```

- 其中上面文本图表中的SRAM Hit为当前层Tensor所占用的SRAM大小，表示与未开启SRAM对比，会节省对应大小的DDR读写数据量。
- Predict SRAM Hit RW Amount为整个模型SRAM的读写预测情况，表示与未开启SRAM对比，每帧可节省的DDR读写数据量。
- 注意：Linux环境日志重定向到终端，Android环境日志重定向到logcat。

5.8 模型剪枝

模型剪枝一般可分为有损剪枝和无损剪枝两种方式，RKNN-Toolkit2的模型剪枝为无损剪枝，也就是可以在减小模型大小和计算量的前提下，不会降低模型的浮点精度，甚至可以提高模型的量化精度。

但并不是所有模型都可以进行无损剪枝，无损剪枝是根据模型的权重的稀疏化程度，来去除一些对模型结果不造成影响的权重和Feature通道，以减小模型的大小和模型的计算量。在启用模型剪枝配置后（将`rknn.config()`的`model_pruning`参数设为`True`），模型转换时会自动根据权重的稀疏化程度对模型进行剪枝。

如果模型剪枝成功，则会打印剪枝结果，如下：

```
I model_pruning results:

I Weight: -1.12145 MB (-6.9%)

I GFLOPs: -0.15563 (-13.4%)
```

其中，Weight剪掉了1.12145 MB（占比6.9%），模型运算量减少0.15563 GFLOPs（占比13.4%）。

如果因模型稀疏化程度不够而剪枝失败，则不会做任何处理（也不会打印上述信息），因此该配置并不会影响正常的模型转换。

5.9 模型加密

模型加密指的是生成完RKNN模型后，再重新对其做进一步处理。使用`rknn.export_rknn()`生成的模型，可以通过Netron等第三方工具查看图结构。模型加密后，Netron等第三方工具将无法查看相应的网络结构，也无法获取权重，起到对模型的保护作用。当前加密后的RKNN模型使用方法和未加密的模型一样，不需要在开发板推理时做任何修改。

使用方法如下：

```
# Create RKNN object
rknn = RKNN()
# Export encrypted RKNN model
crypt_level= 1
ret = rknn.export_encrypted_rknn_model("input.rknn", "encrypt.rknn", crypt_level)
if ret != 0:
    print('Encrypt RKNN model failed!')
```

`crypt_level`用来指定加密等级，有1，2和3三个等级。默认值为1。等级越高，安全性越高，解密越耗时；反之，安全性越低，解密越快。

- 支持平台：RK3562/RK3566/RK3568/RK3576/RK3588/RV1126B

5.10 Cacheable内存一致性

Cacheable内存一致性问题是指当CPU和NPU设备都会访问同一块带cache标志的内存时，CPU会将数据缓存到cache中，如果NPU访问到的DDR数据与CPU cache不一致，会导致读取数据错误。因此要调用刷新cache的接口保证CPU和NPU访问到的DDR内存数据是一致的。本章节介绍了同步数据的方向以及如何使用`rknn_mem_sync`接口刷新cache。

5.10.1 Cacheable内存同步的方向

当CPU写数据到cacheable的内存，之后NPU访问该内存时，要保证CPU cache的数据同步到DDR中，此时同步的方向是指从CPU到NPU设备；当NPU写完数据，CPU开始访问该内存时，要保证DDR的数据与CPU cache中的一致，此时同步的方向是指从NPU设备到CPU。RKNN C API提供了`rknn_mem_sync_mode`枚举类型表示cacheable内存同步的方向，数据结构如下：

```
/*
 The mode to sync cacheable rknn memory.
*/

typedef enum _rknn_mem_sync_mode {

RKNN_MEMORY_SYNC_TO_DEVICE = 0x1, /* the mode used for consistency of device access after
CPU accesses data. */

RKNN_MEMORY_SYNC_FROM_DEVICE = 0x2, /* the mode used for consistency of CPU access after
device accesses data. */
```

```
RKNN_MEMORY_SYNC_SYNC_BIDIRECTIONAL =
    RKNN_MEMORY_SYNC_TO_DEVICE | RKNN_MEMORY_SYNC_FROM_DEVICE, /* the mode used for
consistency of data access
between device and CPU in both directions. */

} rknn_mem_sync_mode;
```

- RKNN_MEMORY_SYNC_TO_DEVICE： 表示数据同步方向是CPU到NPU设备
- RKNN_MEMORY_SYNC_FROM_DEVICE： 表示数据同步方向是NPU设备到CPU
- RKNN_MEMORY_SYNC_SYNC_BIDIRECTIONAL：表示数据在NPU和CPU之间双向同步在用户不确定同步数据的方向时可以使用该枚举

在明确数据同步方向时，建议使用单方向刷新CPU cache模式，能避免多余的刷新CPU cache动作导致性能损耗。

5.10.2 同步Cacheable内存

明确数据同步方向之后，就可以使用`rknn_mem_sync`接口对cacheable内存做同步。接口形式如下：

```
int rknn_mem_sync(rknn_context context, rknn_tensor_mem* mem, rknn_mem_sync_mode mode);
```

其中，context是上下文(在非RV1103系列/RV1106系列平台上默认设置为NULL)，mem是`rknn_create_mem`接口返回的`rknn_tensor_mem*`指针类型，mode是指同步数据的方向。

5.11 模型稀疏化推理

该功能用于Torch模型在训练阶段对模型权重自动稀疏化并使用RKNN进行稀疏化推理，提高模型推理速度。目前该功能仅支持RK3576、RK1126B。

5.11.1 稀疏化原理

模型权重稀疏化在训练阶段根据用户自定义的方式，进行权重置零操作，具体方式有4:2输入方向稀疏化，4:2输出方向稀疏化，16:4输入输出稀疏化，16:4输出输入稀疏化。其中4:2输入方向稀疏化实现原理如图5-16所示，将模型权重沿输入方向，在连续的4个数值中选择两个置零。需要注意的是，当模型权重指定方向非4对齐时，会进行补齐操作。

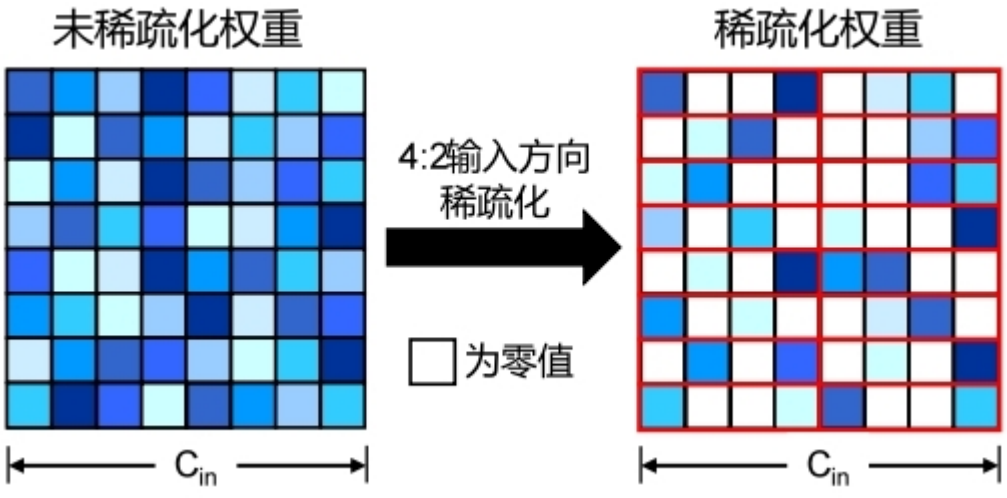


图5-16 4:2 输入方向稀疏化原理图

有关四种权重稀疏化说明如表5-16所示，其中输入输出方向为权重的方向，例如2维卷积shape为C_{out}×C_{in}×K_h×K_w，输入方向即C_{in}，输出方向即C_{out}。在稀疏率为75%的方式中前置训练模型可以是未稀疏化模型也可以是稀疏模型，建议使用单方向稀疏模型作为前置预训练模型，可减少稀疏化后的精度损失。

表5-10 模型权重稀疏化说明

稀疏化方式	稀疏率	说明
4:2输入方向稀疏化	50%	沿输入方向对权重稀疏化
4:2输出方向稀疏化	50%	沿输出方向对权重稀疏化
16:4输入输出稀疏化	75%	先沿输入方向4:2稀疏，再沿输出方向4:2稀疏
16:4输出输入稀疏化	75%	先沿输出方向4:2稀疏，再沿输入方向4:2稀疏

5.11.2 训练稀疏化模型

1.首先确保cuda可使用并安装对应python版本的autosparsity包

```
pip install autosparsity-1.0-cp38-cp38m-linux_x86_64.whl
```

autosparsity安装包路径为：https://github.com/airockchip/rknn-toolkit2/tree/master/autosparsity/packages

2.以torchvision中的resnet50的4:2 输入方向稀疏化为例，进行权重稀疏化训练

```
import torch
import torchvision.models as models
from autosparsity.sparsity import sparsity_model

if __name__ == "__main__":

    model = models.resnet50(pretrained=True).cuda()
    optimizer = None
    mode = 0
    sparsity_model(model, optimizer, mode)
    model.eval()
    x = torch.randn((1,3,224,224)).cuda()
    torch.onnx.export(
        model, x, 'resnet50.onnx', input_names=['inputs'], output_names=['outputs']
    )
```

自定义模型的稀疏化在模型训练之前添加 `sparsity_model` 函数即可，参考示例如下：

```
# insert model autosparsity code before training
    import torch
    import torchvision.models as models
    from autosparsity.sparsity import sparsity_model

    ...

    model = models.resnet34(pretrained=True).cuda()
```

```
    mode = 0
    sparsity_model(model, optimizer, mode)

# normal training
    x, y = DataLoader(args)
    for epoch in range(epochs):
        y_pred = model(x)
        loss = loss_func(y_pred, y)
        loss.backward()
        optimizer.step()
    ...
```

注：Attention 模型稀疏化训练依赖于一个已训练好的前置模型。所以在做稀疏化训练之前，请先训练一个效果'不错'的模型，在该模型基础上进行稀疏化训练。

有关`sparsity_model`函数的参数说明如下：

表5-11 sparsity_model函数各参数说明

参数	详细说明
model	原训练模型
optimizer	原优化器，默认为None
mode	稀疏化方式，可选值为0, 1, 2, 3，默认为0 0: 4:2输入方向稀疏化(50%稀疏率) 1: 4:2输出方向稀疏化(50%稀疏率) 2: 16:4输入输出稀疏化(75%稀疏率) 3: 16:4输出输入稀疏化(75%稀疏率)
verbose	log等级，可选值为0, 1, 2, 3，默认为2 0: Errors 1: Errors and Warnings 2: Errors, warnings and info
whitelist	稀疏化支持的module列表，支持1d conv，2d conv，3d conv, linear, MultiheadAttention，默认[torch.nn.Linear, torch.nn.Conv2d]
allowed_layer_names	允许稀疏化的层名，用户配置时则只稀疏指定层，默认None
disallowed_layer_names	不允许稀疏化的层名，用户配置时则会跳过该层，默认[]
fast	设为True代表使用快速方法计算mask，默认为False(默认的mask计算方法针对部分模型会失效，若稀疏化报错可尝试将该参数设为True)

5.11.3 RKNN稀疏化推理使用方法

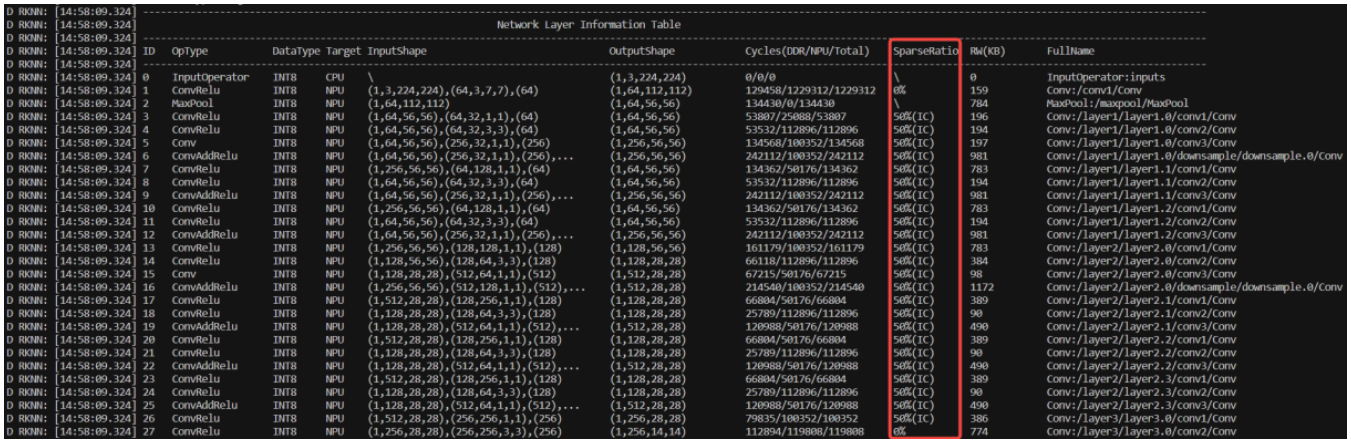
通过RKNN-Toolkit2中`config()`接口的"sparse_infer"参数设置模型稀疏化的开启和关闭,对应的参数为`True/False`，默认值为`False`。开启稀疏化推理的参考代码如下:

```
rknn.config(target_platform='rk3576', sparse_infer=True)
```

完整的稀疏化Python推理代码可参考：https://github.com/airockchip/rknn-toolkit2/tree/master/autosparsity/examples

使用C API进行部署时，首先使用RKNN-Toolkit2在`config()`接口中设置“`sparse_infer`”参数为True生成带稀疏化推理的RKNN模型，之后正常调用通用API接口流程或零拷贝接口流程即可。

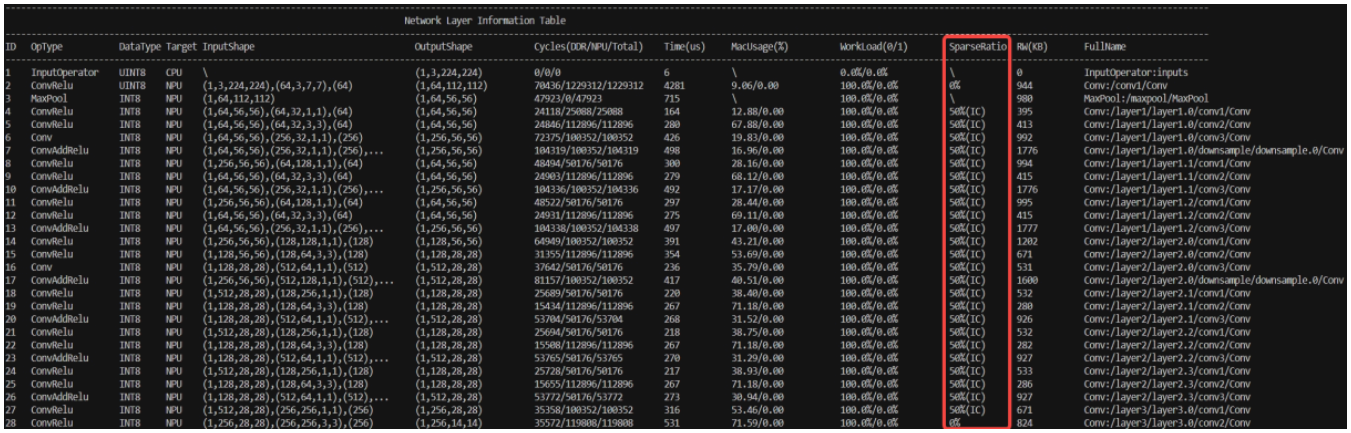
使用Python代码推理时可在构建RKNN对象时设置`verbose=True`，开启日志打印各层稀疏化情况；使用C API推理时通过设置环境变量`RKNN_LOG_LEVEL=4`，开启日志打印各层稀疏化情况。日志信息如下：



D RKNN: [14:58:09.324]
D RKNN: [14:58:09.324] Network Layer Information Table
D RKNN: [14:58:09.324]

ID	OpType	DataType	Target	InputShape	OutputShape	Cycles(DDR/NPU/Total)	SparseRatio	RW(KB)	FullName
0	InputOperator	INT8	CPU	\	(1,3,224,224)	0/0/0	\	0	InputOperator:inputs
1	ConvRelu	INT8	NPU	(1,3,224,224),(64,3,7,7),(64)	(1,64,112,112)	129458/1229312/1229312	0%	159	Conv:/conv1/Conv
2	MaxPool	INT8	NPU	(1,64,112,112)	(1,64,56,56)	134430/0/134430	\	784	MaxPool:/maxpool/MaxPool
3	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,1,1),(64)	(1,64,56,56)	53807/25088/53807	50%(IC)	196	Conv:/layer1/layer1.0/conv1/Conv
4	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	53532/112896/112896	50%(IC)	194	Conv:/layer1/layer1.0/conv2/Conv
5	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	134568/100352/134568	50%(IC)	197	Conv:/layer1/layer1.0/conv3/Conv
6	ConvAddRelu	INT8	NPU	(1,64,56,56),(256,32,1,1),(256),...	(1,256,56,56)	242112/100352/242112	50%(IC)	981	Conv:/layer1/layer1.0/downsample/downsample.0/Conv
7	ConvRelu	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	134362/50176/134362	50%(IC)	783	Conv:/layer1/layer1.1/conv1/Conv
8	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	53532/112896/112896	50%(IC)	194	Conv:/layer1/layer1.1/conv2/Conv
9	ConvAddRelu	INT8	NPU	(1,64,56,56),(256,32,1,1),(256),...	(1,256,56,56)	242112/100352/242112	50%(IC)	981	Conv:/layer1/layer1.1/conv3/Conv
10	ConvRelu	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	134362/50176/134362	50%(IC)	783	Conv:/layer1/layer1.2/conv1/Conv
11	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	53532/112896/112896	50%(IC)	194	Conv:/layer1/layer1.2/conv2/Conv
12	ConvAddRelu	INT8	NPU	(1,64,56,56),(256,32,1,1),(256),...	(1,256,56,56)	242112/100352/242112	50%(IC)	981	Conv:/layer1/layer1.2/conv3/Conv
13	ConvRelu	INT8	NPU	(1,256,56,56),(128,128,1,1),(128)	(1,128,56,56)	161179/100352/161179	50%(IC)	783	Conv:/layer2/layer2.0/conv1/Conv
14	ConvRelu	INT8	NPU	(1,128,56,56),(128,64,3,3),(128)	(1,128,28,28)	66118/112896/112896	50%(IC)	384	Conv:/layer2/layer2.0/conv2/Conv
15	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	67215/50176/67215	50%(IC)	98	Conv:/layer2/layer2.0/conv3/Conv
16	ConvAddRelu	INT8	NPU	(1,256,56,56),(512,128,1,1),(512),...	(1,512,28,28)	214540/100352/214540	50%(IC)	1172	Conv:/layer2/layer2.0/downsample/downsample.0/Conv
17	ConvRelu	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	66804/50176/66804	50%(IC)	389	Conv:/layer2/layer2.1/conv1/Conv
18	ConvRelu	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	25789/112896/112896	50%(IC)	90	Conv:/layer2/layer2.1/conv2/Conv
19	ConvAddRelu	INT8	NPU	(1,128,28,28),(512,64,1,1),(512),...	(1,512,28,28)	120988/50176/120988	50%(IC)	490	Conv:/layer2/layer2.1/conv3/Conv
20	ConvRelu	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	66804/50176/66804	50%(IC)	389	Conv:/layer2/layer2.2/conv1/Conv
21	ConvRelu	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	25789/112896/112896	50%(IC)	90	Conv:/layer2/layer2.2/conv2/Conv
22	ConvAddRelu	INT8	NPU	(1,128,28,28),(512,64,1,1),(512),...	(1,512,28,28)	120988/50176/120988	50%(IC)	490	Conv:/layer2/layer2.2/conv3/Conv
23	ConvRelu	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	66804/50176/66804	50%(IC)	389	Conv:/layer2/layer2.3/conv1/Conv
24	ConvRelu	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	25789/112896/112896	50%(IC)	90	Conv:/layer2/layer2.3/conv2/Conv
25	ConvAddRelu	INT8	NPU	(1,128,28,28),(512,64,1,1),(512),...	(1,512,28,28)	120988/50176/120988	50%(IC)	490	Conv:/layer2/layer2.3/conv3/Conv
26	ConvRelu	INT8	NPU	(1,512,28,28),(256,256,1,1),(256)	(1,256,28,28)	79835/100352/100352	50%(IC)	386	Conv:/layer3/layer3.0/conv1/Conv
27	ConvRelu	INT8	NPU	(1,256,28,28),(256,256,3,3),(256)	(1,256,14,14)	112894/119808/119808	0%	774	Conv:/layer3/layer3.0/conv2/Conv

图5-17 4:2 输入方向稀疏化Python打印日志



Network Layer Information Table

ID	OpType	DataType	Target	InputShape	OutputShape	Cycles(DDR/NPU/Total)	Time(us)	MacUsage(%)	WorkLoad(0/1)	SparseRatio	RW(KB)	FullName
1	InputOperator	UINT8	CPU	\	(1,3,224,224)	0/0/0	6	\	0.0%/0.0%	\	0	InputOperator:inputs
2	ConvRelu	UINT8	NPU	(1,3,224,224),(64,3,7,7),(64)	(1,64,112,112)	70436/1229312/1229312	4281	9.06/0.00	100.0%/0.0%	0%	944	Conv:/conv1/Conv
3	MaxPool	INT8	NPU	(1,64,112,112)	(1,64,56,56)	47923/0/47923	715	\	100.0%/0.0%	\	980	MaxPool:/maxpool/MaxPool
4	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,1,1),(64)	(1,64,56,56)	24118/25088/25088	164	12.88/0.00	100.0%/0.0%	50%(IC)	395	Conv:/layer1/layer1.0/conv1/Conv
5	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	24846/112896/112896	280	67.88/0.00	100.0%/0.0%	50%(IC)	413	Conv:/layer1/layer1.0/conv2/Conv
6	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	72375/100352/100352	426	19.83/0.00	100.0%/0.0%	50%(IC)	992	Conv:/layer1/layer1.0/conv3/Conv
7	ConvAddRelu	INT8	NPU	(1,64,56,56),(256,32,1,1),(256),...	(1,256,56,56)	104319/100352/104319	498	16.96/0.00	100.0%/0.0%	50%(IC)	1776	Conv:/layer1/layer1.0/downsample/downsample.0/Conv
8	ConvRelu	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	48494/50176/50176	300	28.16/0.00	100.0%/0.0%	50%(IC)	994	Conv:/layer1/layer1.1/conv1/Conv
9	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	24903/112896/112896	279	68.12/0.00	100.0%/0.0%	50%(IC)	415	Conv:/layer1/layer1.1/conv2/Conv
10	ConvAddRelu	INT8	NPU	(1,64,56,56),(256,32,1,1),(256),...	(1,256,56,56)	104336/100352/104336	492	17.17/0.00	100.0%/0.0%	50%(IC)	1776	Conv:/layer1/layer1.1/conv3/Conv
11	ConvRelu	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	48522/50176/50176	297	28.44/0.00	100.0%/0.0%	50%(IC)	995	Conv:/layer1/layer1.2/conv1/Conv
12	ConvRelu	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	24931/112896/112896	275	69.11/0.00	100.0%/0.0%	50%(IC)	415	Conv:/layer1/layer1.2/conv2/Conv
13	ConvAddRelu	INT8	NPU	(1,64,56,56),(256,32,1,1),(256),...	(1,256,56,56)	104338/100352/104338	407	17.00/0.00	100.0%/0.0%	50%(IC)	1777	Conv:/layer1/layer1.2/conv3/Conv
14	ConvRelu	INT8	NPU	(1,256,56,56),(128,128,1,1),(128)	(1,128,56,56)	64949/100352/100352	391	43.21/0.00	100.0%/0.0%	50%(IC)	1202	Conv:/layer2/layer2.0/conv1/Conv
15	ConvRelu	INT8	NPU	(1,128,56,56),(128,64,3,3),(128)	(1,128,28,28)	31355/112896/112896	354	53.69/0.00	100.0%/0.0%	50%(IC)	671	Conv:/layer2/layer2.0/conv2/Conv
16	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	37642/50176/50176	236	35.79/0.00	100.0%/0.0%	50%(IC)	531	Conv:/layer2/layer2.0/conv3/Conv
17	ConvAddRelu	INT8	NPU	(1,256,56,56),(512,128,1,1),(512),...	(1,512,28,28)	81157/100352/100352	417	40.51/0.00	100.0%/0.0%	50%(IC)	1600	Conv:/layer2/layer2.0/downsample/downsample.0/Conv
18	ConvRelu	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	25689/50176/50176	220	38.40/0.00	100.0%/0.0%	50%(IC)	532	Conv:/layer2/layer2.1/conv1/Conv
19	ConvRelu	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	15434/112896/112896	267	71.18/0.00	100.0%/0.0%	50%(IC)	280	Conv:/layer2/layer2.1/conv2/Conv
20	ConvAddRelu	INT8	NPU	(1,128,28,28),(512,64,1,1),(512),...	(1,512,28,28)	53704/50176/53704	268	31.52/0.00	100.0%/0.0%	50%(IC)	926	Conv:/layer2/layer2.1/conv3/Conv
21	ConvRelu	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	25694/50176/50176	218	38.75/0.00	100.0%/0.0%	50%(IC)	532	Conv:/layer2/layer2.2/conv1/Conv
22	ConvRelu	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	15508/112896/112896	267	71.18/0.00	100.0%/0.0%	50%(IC)	282	Conv:/layer2/layer2.2/conv2/Conv
23	ConvAddRelu	INT8	NPU	(1,128,28,28),(512,64,1,1),(512),...	(1,512,28,28)	53765/50176/53765	270	31.29/0.00	100.0%/0.0%	50%(IC)	927	Conv:/layer2/layer2.2/conv3/Conv
24	ConvRelu	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	25728/50176/50176	217	38.93/0.00	100.0%/0.0%	50%(IC)	533	Conv:/layer2/layer2.3/conv1/Conv
25	ConvRelu	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	15655/112896/112896	267	71.18/0.00	100.0%/0.0%	50%(IC)	286	Conv:/layer2/layer2.3/conv2/Conv
26	ConvAddRelu	INT8	NPU	(1,128,28,28),(512,64,1,1),(512),...	(1,512,28,28)	53772/50176/53772	273	30.94/0.00	100.0%/0.0%	50%(IC)	927	Conv:/layer2/layer2.3/conv3/Conv
27	ConvRelu	INT8	NPU	(1,512,28,28),(256,256,1,1),(256)	(1,256,28,28)	35358/100352/100352	316	53.46/0.00	100.0%/0.0%	50%(IC)	671	Conv:/layer3/layer3.0/conv1/Conv
28	ConvRelu	INT8	NPU	(1,256,28,28),(256,256,3,3),(256)	(1,256,14,14)	35572/119808/119808	531	71.59/0.00	100.0%/0.0%	0%	824	Conv:/layer3/layer3.0/conv2/Conv

图5-18 4:2 输入方向稀疏化C API打印日志

其中`SparseRation`代表稀疏率，4:2输入方向稀疏化对应50%(IC)，4:2输出方向稀疏化对应50%(OC)，16:4输入输出稀疏化和16:4输出输入稀疏化对应75%。0%则代表未做稀疏化。

5.11.4 RKNN稀疏化推理限制

稀疏化推理是基于NPU的硬件架构实现，受硬件规格限制，RK3576的稀疏化推理目前只支持2d Conv并且group参数为1，其余限制如下：

表5-12 RKNN稀疏化推理限制

	通道数量	数据类型
4:2输入方向稀疏化	输入32对齐，输出32对齐	仅支持Int8 不支持Float16
4:2输出方向稀疏化	输入32对齐，输出32对齐	仅支持Int8 不支持Float16
16:4输入输出稀疏化	输入32对齐，输出32对齐	仅支持Int8 不支持Float16
16:4输出输入稀疏化	输入32对齐，输出32对齐	仅支持Int8 不支持Float16

5.12 生成部署C代码

RKNN-Toolkit2 2.0.0版本新增 Codegen 接口，用于生成模型部署代码，简化开发者的上手难度。Codegen 基于 CAPI 零拷贝接口进行二次封装，接口风格与 RKNN_Model_Zoo 中的 demo 一致。生成的部署代码可用于直接测试性能、验证精度，开发者也可以基于生成的代码进行二次开发。

使用示例：

```
ret = rknn.codegen(output_path='./rknn_app_demo',
                   inputs=['../../caffe/mobilenet_v2/dog_224x224.jpg'], overwrite=True)
```

- 调用`codegen`接口前，必须先调用`rknn.export`接口保存RKNN模型。
- `output_path`为输出文件夹目录，用户可配置目录名称。
- `inputs`填写模型输入的路径列表，允许不填。有效文件格式为`jpg`/`png`/`npy`，以`npy`文件为输入时，`npy`数据的维度信息应与模型输入的维度信息保持一致。
- `overwrite`设为`True`时，会覆盖`output_path`指定目录下的文件。默认值为`False`。
- 生成部署代码后，请参考生成目录下的`README.md`文档说明进行编译、测试。
- 若`inputs`填入有效值，部署代码示例在推理后，会评估C API接口与RKNN-Toolkit2模拟器之间的推理结果差异，评估方式为对比每一个输出的余弦相似度。
- 无NPU硬件平台限制，要求板端系统为Linux或Android。
- 支持量化、非量化模型。

5.13 ONNX模型编辑

部分模型在转RKNN模型后，可能存在冗余的op，常见于模型的输入输出节点处存在冗余的reshape、transpose op，影响了RKNN模型的推理性能。RKNN-Toolkit2提供了`onnx_edit`接口，用于修改ONNX模型的输入输出的维度定义，使调整后的onnx模型能转出性能更好的RKNN模型，减少冗余的reshape、transpose op。

5.13.1 onnx_edit接口说明

使用示例：

```
from rknn.utils import onnx_edit
ret = onnx_edit(model = './concat_block.onnx',
    export_path = './concat_block_edited.onnx',
    inputs_transform = {'k_in': 'a,b,c,d->1,ad,b,c'},
    outputs_transform = {'k_cache': 'a,b,c,d->1,ab,c,d'},
    dataset = './dataset.txt'
)
```

- model: 填入待修改模型，为必填参数
- export_path: 填入新模型的生成路径，为必填参数
- inputs_transform: 填入输入节点的变换公式字典，`key`为节点名称，`value`为变换公式。为可选参数，默认为空字典
- outputs_transform: 填入输出节点的变换公式字典。为可选参数，默认为空字典

- dataset: 填入输入数据的路径集文件，文件格式要求与`rknn.build`接口对`dataset`的要求一致。填入后，`onnx_edit`接口除了对模型的输入输出定义进行调整，也会将`dataset`中对应的数据进行调整，在`export_path`同级目录下生成新的`dataset`数据，可用于新模型的验证、量化。为可选参数，默认为空。

5.13.2 onnx_edit变换公式说明

onnx_edit接口中的`inputs_transform`、`outputs_transform`需要填入变换公式。变换公式的定义与`einsum`算子的定义类似，例如'`a,b,c,d->1,ad,b,c`'公式，指将原始的维度为'`a,b,c,d`'，变换后为'`1,ad,b,c`'。变换公式的填写规则如下：

- 必须有两部分字符组成，并用 '->' 隔开，左边为原始字符shape，右边是变换后字符shape
- 字符shape里面的符号只允许是 [a-z] 或 ',' '1'，除了'1'以外，不支持其他数字shape字符，原因是相同数字字符无法判断 transpose 的前后关系
- 字符[a-z]认为是独立的，没有顺序关系，即 'a,b->b,a' 和 'c,a->a,c' 表示的是一样的变换
- 等式左边的原始字符shape，被 ',' 分隔成 n 份，n的个数必须和模型的维度匹配，例如模型输入定义是 [32,4,1,64], 输入字符可以是 '`a,b,c,d`' 或 '`a,b,1,d`'
- 原始字符shape的每一个符号，除了'1'，都必须存在于变换后字符shape，例如 '`a,1,c,d->ac,d,1`' 是有效的，'`a,1,c,d->ac,1`' 是无效的，缺了 'd'
- 变换后字符shape，允许插入任意个数的 '1'，达成扩维效果，例如 '`a,b,c-> a,1,cb,1,1`'
- 原始字符shape，允许用多个字母以及赋值公式来表示对shape进行拆分，例如原始输入定义是 [32,4,1,64]，'`ab,c,d,qk[a=2,k=8]->aq,cd,1,kb`'，表示将 32 拆分成 2x16，将 64 拆分成 8x8，再进行 transpose, reshape 操作。其中 '[ ]' 的部分称为赋值公式，多个公式用 ',' 符号分隔。此外，允许拆分中的某个字符没有赋值，此时会自动推断对应的shape，例如赋值公式只给了 a=2，已知在模型中 ab=32，则自动推断出 b=16；若推断出的shape异常会直接报错，比如 ab=32，若赋值 a=5，则 b=6.4，又维度必须是整数，此时会抛出异常错误。

5.13.3 变换公式示例

- 将3维输入修改为4维输入：'a,b,c->a,b,1,c'
- 将5维输入修改为4维输入：'a,b,c,d,e->ab,c,d,e'
- 进行transpose(0,3,1,2)操作：'a,b,c,d->a,d,b,c'
- Transpose并合并部分维度：'a,b,c,d->d,acb,1'
- 拆分维度、transpose、合并维度：'a,bc,de,f[b=2,d=4]->ab,fe,dc,1'

6 量化说明

6.1 量化介绍

6.1.1 量化定义

模型量化是指将深度学习模型中的浮点参数和操作转换为定点表示，如FLOAT32转换为INT8等。量化能够降低内存占用，实现模型压缩和推理加速，但会造成一定程度的精度损失。

6.1.2 量化计算原理

以线性非对称量化为例，浮点数量化为有符号定点数的计算原理如下：

$$x_{int} = clamp(\lfloor \frac{x}{s} \rceil + z; -2^{b-1}, 2^{b-1} - 1) \tag{6-1}$$

其中x为浮点数，x_{int}为量化定点数，$\lfloor \cdot \rceil$为四舍五入运算，s为量化比例因子，z为量化零点，b为量化位宽，如INT8数据类型中b为8；$clamp$为截断运算，具体定义如下：

$$clamp(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \le x \le c, \\ c, & x > c, \end{cases} \tag{6-2}$$

从定点数转换为浮点数称为反量化过程，具体定义如下：

$$x \approx \widehat{x} = s(x_{int} - z) \tag{6-3}$$

设量化范围为$(q_{\min})$，截断范围为$(c_{\min})$，量化参数s和z的计算公式如下：

$$s = \frac{q_{\max} - q_{\min}}{c_{\max} - c_{\min}} = \frac{q_{\max} - q_{\min}}{2^b - 1} \tag{6-4}$$

$$z = c_{\max} - \lfloor \frac{q_{\max}}{s} \rceil \quad 或 \; z = c_{\min} - \lfloor \frac{q_{\min}}{s} \rceil \tag{6-5}$$

其中截断范围是根据量化的数据类型决定，例如INT8的截断范围为(-128, 127)；量化范围根据不同的量化算法确定，具体可参考6.1.6量化算法章节。

6.1.3 量化误差

量化会造成模型一定程度的精度丢失。根据公式(6-1)可知，量化误差来源于舍入误差和截断误差，即$\lfloor \cdot \rceil$和$clamp$运算。四舍五入的计算方式会产生舍入误差，误差范围为$\left(-\frac{1}{2}s, \frac{1}{2}s\right)$。当浮点数$x$过大，比例因子$s$过小时，容易导致量化定点数超出截断范围，产生截断误差。理论上，比例因子s的增大可以减小截断误差，但会造成舍入误差的增大。因此为了权衡两种误差，需要设计合适的比例因子和零点，来减小量化误差。

6.1.4 线性对称量化和线性非对称量化

线性量化中定点数之间的间隔是均匀的，例如INT8线性量化将量化范围均匀等分为256个数。线性对称量化中零点是根据量化数据类型确定并且零点z位于量化定点数范围上的中心对称点，例如INT8中零点为0。线性非对称量化中零点根据公式(6-5)计算确定并且零点z一般不在量化定点数范围上的中心对称点。

对称量化是非对称量化的简化版本，理论上非对称量化能够更好的处理数据分布不均匀的情况，因此实践中大多采用非对称量化方案。

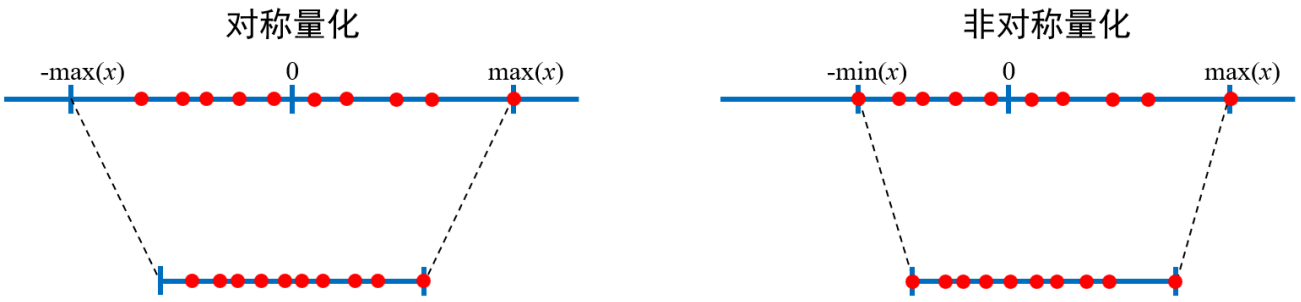


图6-1 线性对称量化和线性非对称量化

6.1.5 Per-Layer量化和Per-Channel量化

Per-Layer量化将网络层的所有通道作为一个整体进行量化，所有通道共享相同的量化参数。Per-Channel量化将网络层的各个通道独立进行量化，每个通道有自己的量化参数。Per-Channel量化更好的保留各通道的信息，能够更好的适应不同通道之间的差异，提供更好的量化效果。

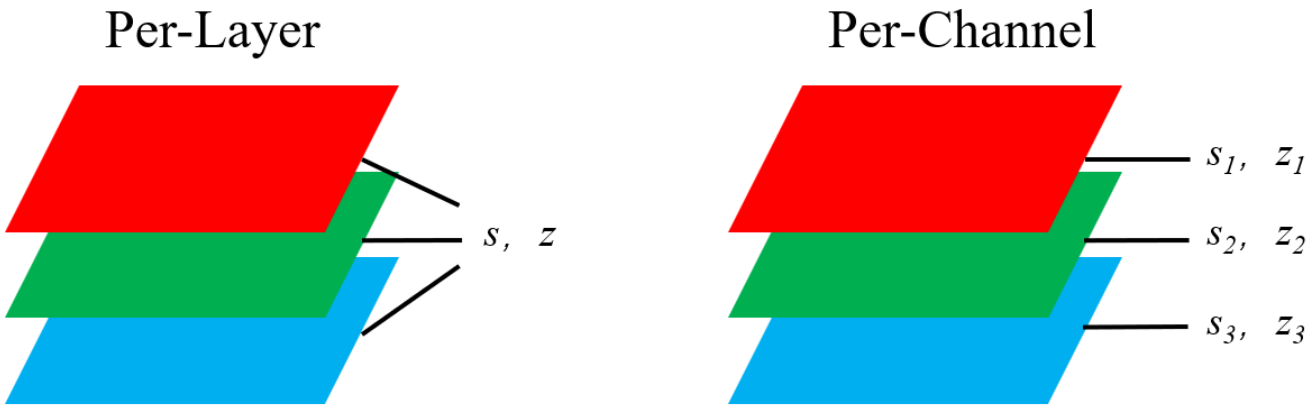


图6-2 Per-Layer量化和Per-Channel量化

注：RKNN-Toolkit2中的Per-Channel量化中只针对权重进行Per-Channel量化，激活值和中间值仍为Per-Layer量化。

6.1.6 量化算法

量化比例因子s和零点z是影响量化误差的关键参数，而量化范围的求解对量化参数起到决定性作用。本章节介绍三种关于量化范围求解的算法，Normal，KL-Divergence和MMSE。

Normal量化算法是通过计算浮点数中的最大值和最小值直接确定量化范围的最大值和最小值。从6.1.2量化计算原理可知，Normal量化算法不会产生截断误差，但对异常值很敏感，因为大异常值可能会导致舍入误差过大。

$$q_{\min} = \min\ \mathbf{V} \tag{6-6}$$

$$q_{\max} = \max\ \mathbf{V} \tag{6-7}$$

其中V为浮点数Tensor。

KL-Divergence量化算法计算浮点数和定点数的分布，通过调整不同的阈值来更新浮点数和定点数的分布，并根据KL散度最小化两个分布的相似性来确定量化范围的最大值和最小值。KL-Divergence量化算法通过最小化浮点数和定点数之间的分布差异，能够更好地适应非均匀的数据分布并缓解少数异常值的影响。

$$\underset{q_{\min},q_{\max}}{\arg\min} H(\Psi(\mathbf{V}),\ \Psi(\mathbf{V_{int}})) \tag{6-8}$$

其中$H(\cdot,\cdot)$为KL散度计算公式，$\Psi(\cdot)$为分布函数，将对应数据计算为离散分布，V_{int}为量化定点数Tensor。

MMSE量化算法通过最小化浮点数与量化反量化后浮点数的均方误差损失，确定量化范围的最大值和最小值，在一定程度上缓解大异常值带来的量化精度丢失问题。由于MMSE量化算法的具体实现是采用暴力迭代搜索近似解，速度较慢，内存开销较大，但通常会比Normal量化算法具有更高的量化精度。

$$\arg\min_{q_{\min},q_{\max}} \left\| \mathbf{V} - \widehat{\mathbf{V}}(q_{\min}, q_{\max}) \right\|_F^2 \tag{6-9}$$

其中$\widehat{\mathbf{V}}(q_{\min}, q_{\max})$为$\mathbf{V}$的量化、反量化形式，$||\cdot||_F$为F范数。

6.2 量化配置

6.2.1 量化数据类型

RKNN-Toolkit2支持的量化数据类型为INT8。

6.2.2 量化算法建议

Normal量化算法运行速度快，适用于一般场景。

KL-Divergence量化算法运行速度略慢于Normal量化算法，对于存在非均匀分布的部分模型能够改善量化精度，部分场景下能够缓解少数异常值造成的量化精度丢失问题。

MMSE量化算法运行速度较慢，内存消耗大，相比KL_Divergence量化算法能够更好的缓解异常值造成的量化精度丢失问题。对于量化友好的模型可尝试使用MMSE量化算法来提高量化精度，因为在多数场景下MMSE量化精度要高于Normal和KL-Divergence量化算法。

默认情况下使用Normal量化算法，当遇到量化精度问题时可尝试使用KL-Divergence和MMSE量化算法。

6.2.3 量化校正集建议

量化校正集用于计算激活值的量化范围，在选择量化校正集时应覆盖模型实际应用场景的不同数据分布，例如对于分类模型，量化校正集应包含实际应用场景中不同类别的图片。一般推荐量化校正集数量为20-200张，可根据量化算法的运行时间适当增减。需要注意的是，增加量化校正集数量会增加量化算法的运行时间但不一定能提高量化精度。

6.2.4 量化配置方法

RKNN-Toolkit2中量化的配置方法在rknn.config()和rknn.build()接口实现。其中量化方法配置由rknn.config()接口实现，量化开关和校正集路径的选择由rknn.build()接口实现。

rknn.config()接口包含以下相关量化配置项：

1. quantized_dtype：选择量化类型，目前仅支持线性非对称的INT8量化，默认为asymmetric_quantized-8。
2. quantized_algorithm：选择量化算法，包括Normal，KL-Divergence和MMSE量化算法。可选值为normal，kl_divergence和mmse，默认为normal。
3. quantized_method：选择Per-Layer和Per-Channel量化。可选值为layer和channel，默认为channel。

rknn.build()接口包含以下相关量化配置项：

1. do_quantization：是否开启量化，默认为False。
2. dataset：量化校正集的路径，默认为空。

目前支持文本文件格式，用户可以把用于校正的图片(jpg或png格式)或npy文件路径放到一个.txt文件中。文本文件里每一行为一条路径信息，如：

```
a.jpg
b.jpg
```

如有多个输入，则每个输入对应的文件用空格隔开，如：

```
a0.jpg a1.jpg
b0.jpg b1.jpg
```

6.3 混合量化

混合量化对模型不同层采用不同的量化数据类型，将不适合量化的层使用较高精度的数据类型表达，以此缓解模型量化精度损失的问题，但混合精度量化会增加额外开销，并且需要用户确定不同层的量化数据类型。

6.3.1 混合量化用法

为了在性能和精度之间做更好的平衡，RKNN-Toolkit2提供了混合量化功能，用户可以通过精度分析的输出结果来手动指定各层是否进行量化。

目前混合量化功能支持如下用法：

1. 将指定的量化层改成非量化层，如用 FLOAT16 进行计算。(因NPU上非量化算力较低，所以推理速度会有一定降低)。
2. 每一层的量化参数也可以进行修改。(量化参数不建议修改)

6.3.2 混合量化使用流程

使用混合量化功能时，具体分四步进行。

1. 加载原始模型，生成量化配置文件、临时模型文件和数据文件。具体的接口调用流程如下：

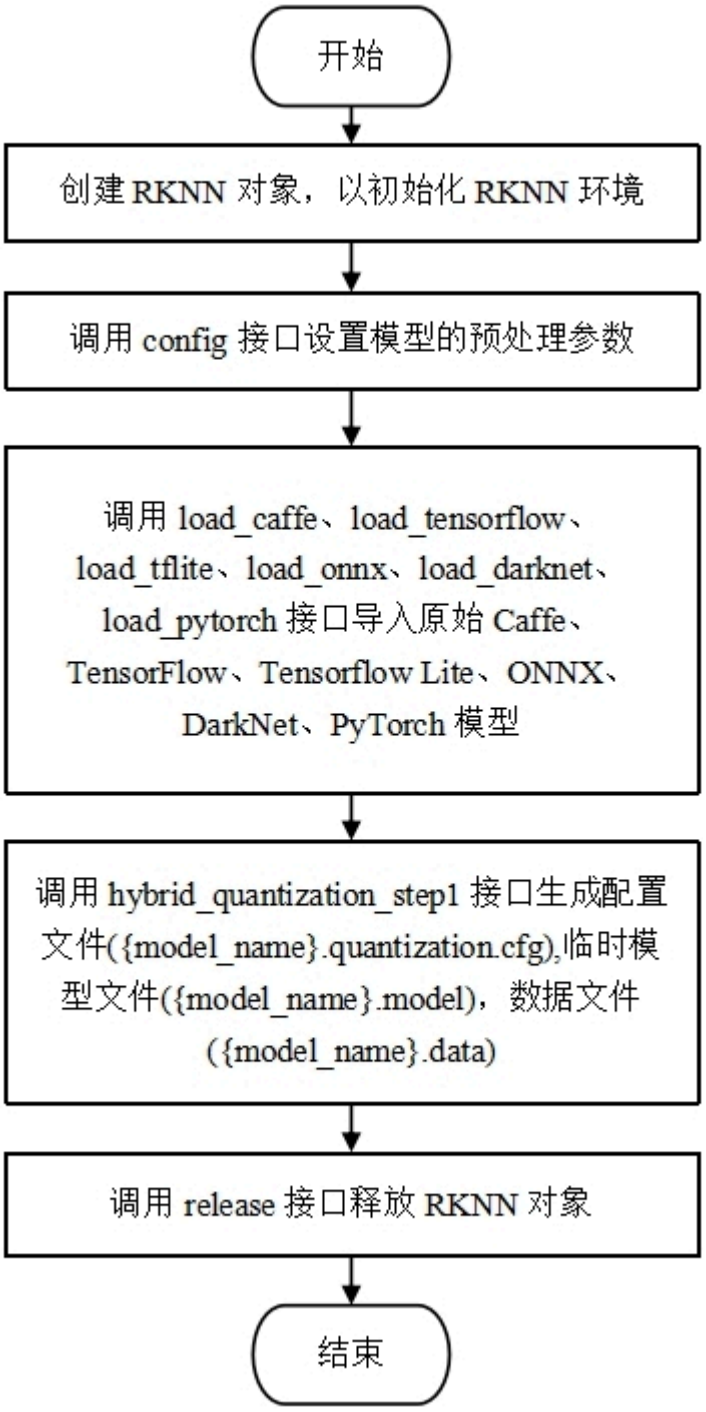


图6-3 混合量化第一步

2. 修改第一步生成的量化配置文件。

第一步调用混合量化接口hybrid_quantization_step1完成后会在当前目录下生成名为{model_name}.quantization.cfg 的配置文件。配置文件格式如下：

```
custom_quantize_layers:
    Conv__350:0: float16
    Conv__358:0: float16
    ....
quantize_parameters:
    ....
    FeatureExtractor/MobilenetV2/Conv/Relu6:0:
        qtype: asymmetric_quantized
```

```
        qmethod: layer
        dtype: int8
        min:
        - 0.0
        max:
        - 6.0
        scale:
        - 0.023529411764705882
        zero_point:
        - -128
        ....
```

custom_quantize_layers下每一行可按照tensor名: 量化类型的格式添加自定义量化层，该tensor对应层的运算类型即改为指定运算类型。目前量化数据类型可选择float16。

quantize_parameters下是模型中每个tensor的量化参数。每个tensor的量化参数按照tensor名: 量化属性和参数的格式呈现。其中min/max代表量化范围的最小最大值，tensor名可根据精度分析输出结果查看或使用Netron打开临时模型文件{model_name}.model查看对应输出tensor名。

3. 生成 RKNN 模型。具体的接口调用流程如下：

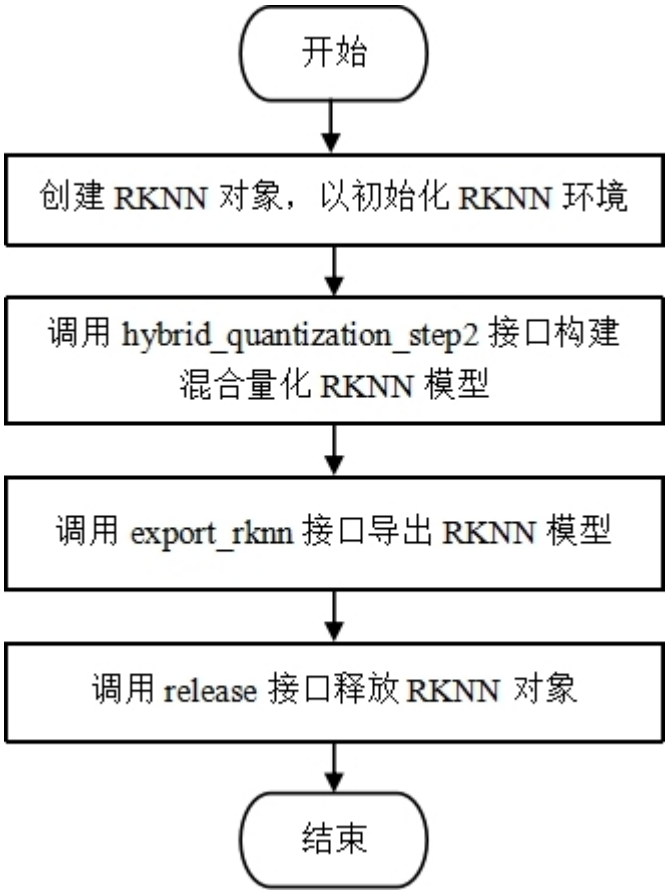


图6-4 混合量化第三步

4. 使用第三步生成的 RKNN 模型进行推理。

注：RKNN-Toolkit2工程中examples/functions/hybrid_quant目录下提供了一个混合量化的例子，具体可以参考该例子对模型进行混合量化。

6.3.3 自动混合量化

为了方便混合量化的使用以及解决模型溢出问题，RKNN-Toolkit2提供了自动混合量化功能。用户可以通过`auto_hybrid`接口，在模型量化时通过欧式距离和余弦距离的阈值来自动调整混合量化模型；在模型非量化时会检查模型逐层是否fp16溢出，将溢出层转换为int16计算。

目前自动混合量化功能支持如下用法：

1. 模型量化时通过欧式距离和余弦距离的阈值来自动调整混合量化模型，目前仅支持从int8量化自动调整为fp16.

```
ret = rknn.build(do_quantization=True, dataset='./dataset.txt', auto_hybrid=True)
```

其中余弦距离和欧式距离的阈值可通过`config`接口中的`auto_hybrid_cos_thresh`和`auto_hybrid_euc_thresh`设置，余弦距离阈值默认为0.98, 欧式距离阈值默认不开启。即默认只对余弦距离精度小于指定阈值的算子转换为fp16，此时欧式距离阈值不起作用。当欧式距离阈值开启时会对小于余弦距离阈值或大于欧式距离阈值的算子转换为fp16。

2. 模型非量化时检查模型逐层是否fp16溢出，将溢出层转换为int16计算，此时需要提供数据集用于计算逐层的数值范围

```
ret = rknn.build(do_quantization=False, dataset='./dataset.txt', auto_hybrid=True)
```

根据给定的输入计算逐层的最大值和最小值，当最大值或最小值超出fp16的上值或下值时会将对应层的算子转换为int16。

6.4 量化感知训练

6.4.1 QAT简介

量化感知训练（英文名称 Quantization-aware Training，简称 QAT）是一种量化训练方式，该方式旨在解决低比特量化的精度损失问题。低比特量化会掉精度，是因为值域从浮点数调整到定点数时会有精度损失，QAT训练时会将量化误差计入训练的损失函数，训练出一个带量化参数的模型。

与 RKNN-Toolkit2 工具提供的后训练量化（英文名称 Post Training Quantization，简称 PTQ）对比，两种量化方法的特点如下:

量化方法	基于原始框架二次训练	数据集	权重参数是否被调整	损失函数	性能
后训练量化（PTQ）	不需要	少量未标注数据	否	无关	最优
量化感知训练（QAT）	需要	完整的训练数据集	是	量化损失计入训练损失函数	存在算子不支持QAT时，性能略弱于 PTQ

6.4.2 QAT原理

量化感知训练时，所有权重的存放格式、算子的计算单元都是按照浮点数进行的，这是为了保证反向传播功能可以正常生效、模型的训练可以正常进行。与训练浮点模型不同，在模型可被量化的位置上，量化感知训练会插入FakeQuantize模块进行伪量化操作，模拟浮点数调整到定点数的精度损失，使其能被损失函数识别、优化，最终使模型转为定点模型时仍可以保持准确的推理结果。

量化感知训练目前已被广泛使用，各主流推理框架皆有实现，可参考下文所列链接获取更详细的使用说明：

Pytorch - https://pytorch.org/blog/quantization-in-practice/#quantization-aware-training-qat

Paddle - https://paddleslim.readthedocs.io/zh-cn/develop/api_cn/dygraph/quanter/qat.html

Tensorflow - https://www.tensorflow.org/model_optimization/guide/quantization/training

6.4.3 QAT使用依据

由于 QAT 需要增加额外的训练代码，且部分开源仓库的代码可能与QAT功能存在冲突，推荐在同时满足以下两种情况时，考虑使用QAT训练进行模型量化:

1. 参考章节7进行量化精度排查，确认RKNN的 PTQ 功能不满足精度要求。
2. 尝试章节6.3进行混合量化，确认RKNN的混合量化功能不满足精度、性能要求。

6.4.4 QAT实现简例及配置说明

以下是各框架 QAT 功能的说明文档，实际使用请以官方文档为准:

Pytorch: https://pytorch.org/docs/stable/quantization.html (Pytorch目前存在多套量化接口，RKNN目前支持 FX 接口生成的量化模型，即 prepare_qat_fx 接口及其配套接口)

Paddle: https://www.paddlepaddle.org.cn/tutorials/projectdetail/3949129#anchor-14

Tensorflow: https://www.tensorflow.org/model_optimization/guide/quantization/training

这里我们以 Pytorch 为例，说明实现 QAT 的流程及一些需要注意的地方。

```
# for 1.10 <= torch <= 1.13
import torch
class M(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Conv(3, 8, 3, 1)
    def forward(self, x):
        x = self.conv(x)
        return x

# initialize a floating point model
float_model = M().train()

from torch.quantization import quantize_fx, QConfig, FakeQuantize,
MovingAverageMinMaxObserver, MovingAveragePerChannelMinMaxObserver

qconfig = QConfig(activation=FakeQuantize.with_args(observer=
            MovingAverageMinMaxObserver,
            quant_min=0,
            quant_max=255,
            reduce_range=False), #reudece_range 默认是True
weight = FakeQuantize.with_args(observer=
            MovingAveragePerChannelMinMaxObserver,
            quant_min=-128,
            quant_max=127,
            dtype=torch.qint8,
            qscheme=torch.per_channel_affine, #参数qscheme默认是torch.per_channel_symmetric
        reduce_range=False))
```

```
qconfig_dict = {"": qconfig}

model_qat = quantize_fx.prepare_qat_fx(float_model, qconfig_dict)

# define the training loop for quantization aware training
def train_loop(model, train_data):
    model.train()

for image, target in data_loader:
    ...

# Run trainin
train_loop(model_qat, train_loop)

model_qat = quantize_fx.convert_fx(model_qat)
```

以上流程代码中，除了qconfig的配置针对RKNN硬件做了部分调整，其余操作均按照官方代码的指引实现。qconfig中主要有以下两处改动：

activation量化配置指定reduce_range为False。reduce_range为False时，有效量化数值范围为-128~127；reduce_range为True时，有效量化数值范围为-64~63，量化效果较差。RKNN 硬件支持reduce_range为False。

weight量化配置指定qscheme为torch.per_channel_affine。默认的torch.per_channel_symmetric会限制zero_point固定为0，RKNN硬件支持zero_point非0，故选择torch.per_channel_affine。

6.4.5 QAT支持的算子

以 Pytorch 为例，使用 QAT 量化的过程中，先对带有权重参数的 Conv, Linear 采取 QAT 规则量化，再检验其他算子，若符合常规量化规则，则对其采用常规量化。在算子不符合 QAT 或常规量化规则的情况下，算子会维持 FP32 的计算规则。

不同框架的支持情况存在差异，用户可以参考以下链接，根据使用的框架及版本，查寻更详细的量化算子支持情况。

Pytorch: https://github.com/pytorch/pytorch/blob/main/torch/ao/quantization/quantization_mappings.py

Paddle: https://github.com/PaddlePaddle/Paddle/blob/86df789a567f1285101c57b6e3ada4b952c58f48/python/paddle/quantization/config.py

Tensorflow: https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/quantization/keras/QuantizeConfig

6.4.6 QAT模型中浮点算子的处理

QAT模型中，当算子无法量化，会采用浮点进行计算。这类算子在转为RKNN模型时，分为两种情况讨论。

1. 前后为可量化算子：

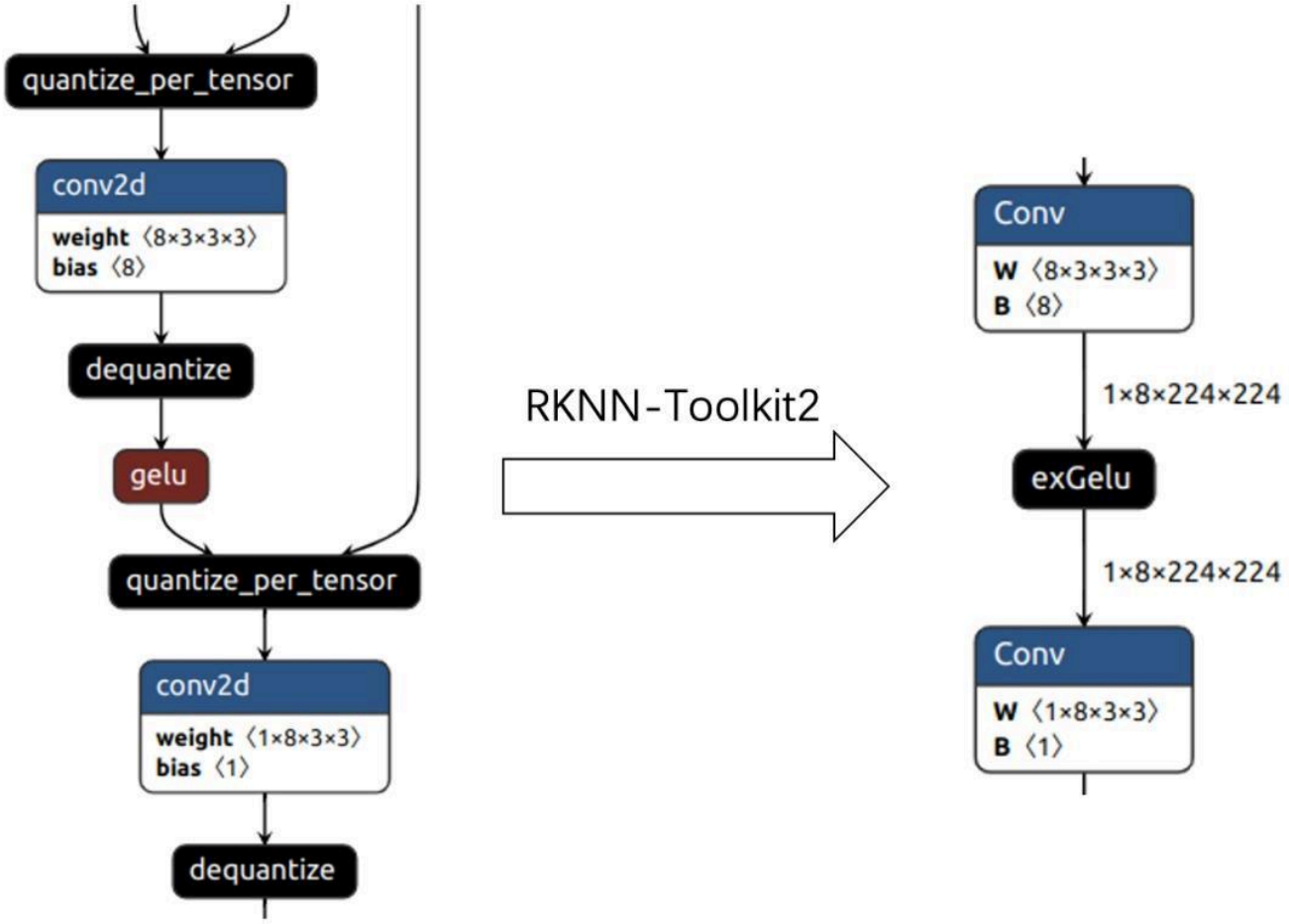


图6-5 QAT OP 前后为可量化算子状态

如上图左边所示，图中的 gelu 算子在原模型中为浮点算子，其前后的 conv 为量化算子，RKNN-Toolkit2 在加载模型时，会将两个已量化算子中间的浮点算子转为量化算子，提升推理性能，这个操作不会影响精度。转为 RKNN 模型后，结构如上图右边所示。

1. 前后存在非量化算子：

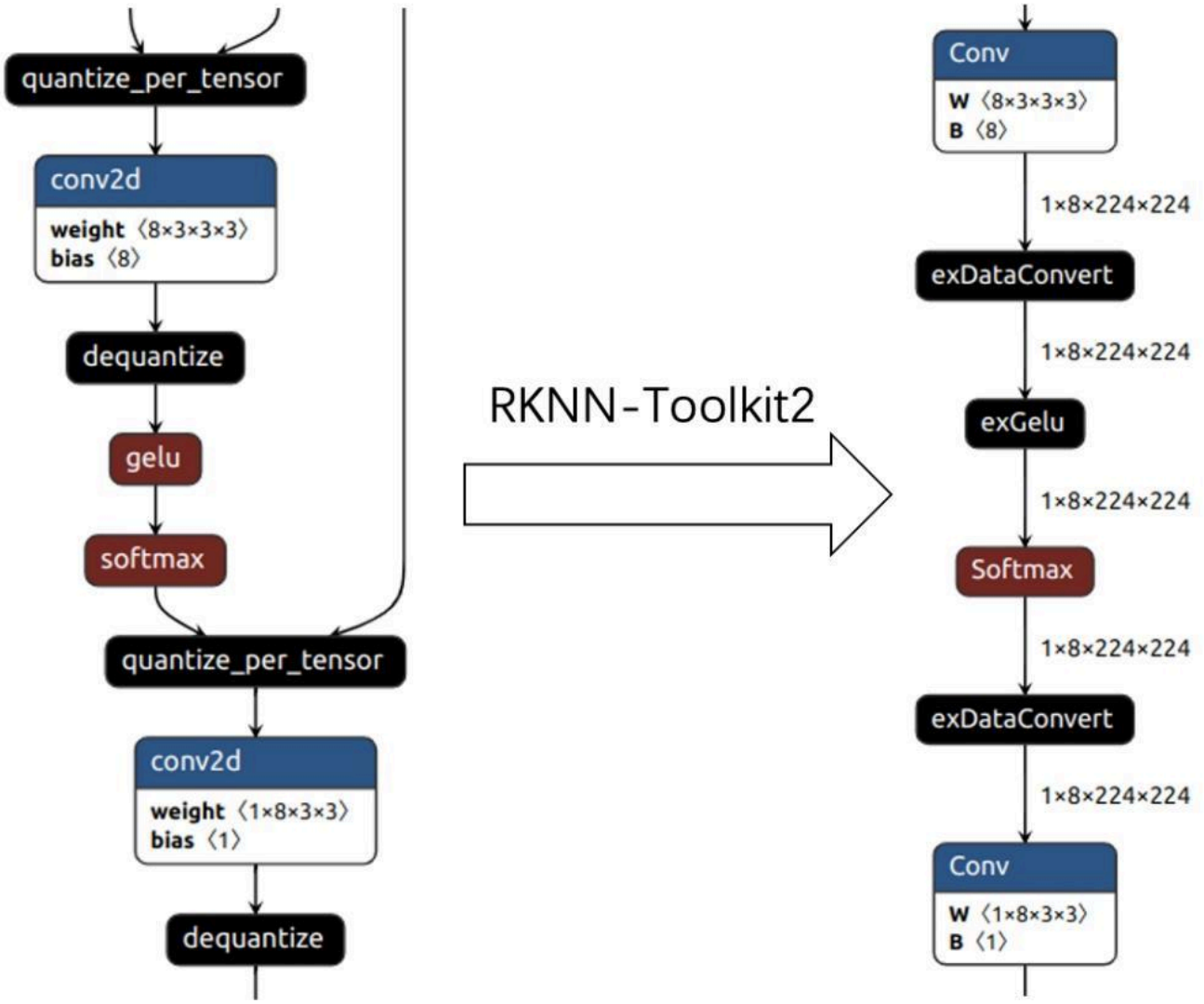


图6-6 QAT OP 前后为非量化算子状态

如上图左边所示，图中的 gelu、softmax 算子在原模型中为浮点算子，其前后的 conv 为量化算子，RKNN-Toolkit2 在加载模型时，由于 gelu、softmax 中间的量化参数缺失，gelu、softmax 仍保持浮点类型。转为 RKNN 模型后结构如上图右边所示，图中 RKNN 模型在 gelu 的前面插入反量化算子，softmax后面插入量化算子，这些插入的量化、反量化算子都会增加额外的耗时。

6.4.7 QAT经验总结

1. QAT 配置

根据不同硬件的特性，QAT训练往往需要调整配置才能达到更好的效果。对于RKNPU而言，建议参考6.4.4的代码说明配置qconfig参数。

2. 模型中保存的量化参数可能需要二次调整

以 sigmoid 为例子，模型中 sigmoid 算子记录的量化参数可能不是实际推理时使用的量化参数。在官方的代码（https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/quantized/cpu/qsigmoid.cpp）中，sigmoid 的量化参数在推理时会进行调整，将 min 置为0，max 置为1。

针对这类算子，RKNN-Toolkit2 在转换 QAT 模型时，会在模型转换阶段调整其量化参数，使推理结果和 Pytorch 原始推理结果更接近。

7 精度排查

模型精度问题排查一般从两个方面进行，一是模拟器精度排查，二是板端Runtime精度排查。模拟器推理结果正确是板端Runtime推理正确的前提，所以需优先保证模拟器推理结果正确，再进行板端Runtime精度问题的排查。

因此本章节将针对**模拟器精度排查**以及**Runtime精度排查**两个方面给出排查建议以及处理方案。下图为具体排查步骤：

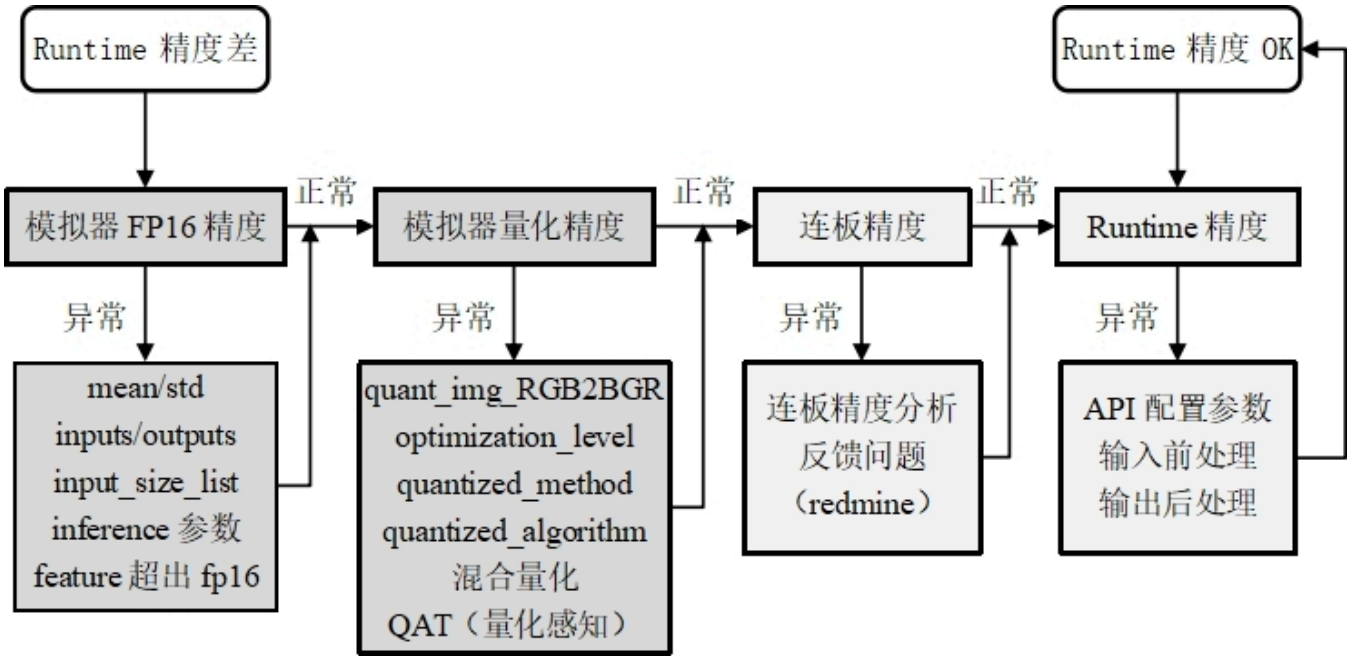


图7-1 精度排查步骤

- 模拟器精度排查，主要分为**模拟器FP16精度**和**模拟器量化精度**排查两个子步骤（上图深色框部分）。
- Runtime精度排查，主要分为**连板精度**和**Runtime精度**排查两个子步骤（上图浅色框部分）。

判断精度可以使用余弦距离作为基本的判断依据，但最终量化对精度的影响仍需要在数据集上验证。

7.1 模拟器精度排查

模拟器推理结果正确是板端Runtime推理正确的前提，所以需优先保证模拟器推理结果正确。

RKNN-Toolkit2上的模拟器推理根据模型是否量化分为**FP16推理**和**量化推理**。FP16推理结果正确是量化推理的结果正确的前提，因此当存在量化推理精度问题时，优先验证FP16推理的正确性，再排查量化推理的精度问题。

7.1.1 模拟器FP16精度

RKNPU目前不支持FP32的计算方式，因此模拟器在不开启量化的情况下，默认是FP16的运算类型，所以只需要在使用rknn.build()接口时，将do_quantization参数设置为False，即可以将原始模型转换为FP16的RKNN模型，接着调用rknn.init_runtime(target=None)和rknn.inference()接口进行FP16模拟推理并获取输出结果。

如果FP16推理输出结果错误，则可以进行以下排查：

- **配置错误**

  模型的配置信息主要集中在rknn.config()接口里，同时在其他的API里也有少数的配置信息可能影响FP16的精度，主要参数如下：

mean_values / std_values：模型的归一化参数，一般原始模型的输入归一化操作是放在模型的输入预处理里实现的，但RKNN模型在推理时可以包含该归一化的操作（在开启量化后，对量化校正集也会先进行归一化操作），因此在原始模型有归一化步骤时，要确保该参数和原始模型使用的归一化参数一致。

input_size_list：rknn.load_tensorflow()、rknn.load_pytorch()和rknn.load_onnx()接口的输入节点shape信息，如果配置错误会导致错误的推理结果。

inputs / outputs：rknn.load_tensorflow()和rknn.load_onnx()接口的输入和输出节点名称，如果配置错误会导致错误的推理结果。

inference接口参数：rknn.inference()接口的输入参数，主要包括 inputs 和 data_format。

- 一般在Python环境下，图像数据都是通过cv2.imread()读取的，此时需要注意cv2.imread()读取的图像格式为BGR，如果原始模型的输入为 BGR（如大部分的caffe模型），则不需要做RGB顺序的调整；而如果原始模型的输入为RGB，则需要调用cv2.cvtColor(img, cv2.COLOR_BGR2RGB)将图像数据转为RGB；另外，通过cv2.imread()读取的图像的shape维度为3维，但是一般模型的输入shape为4维， 因此还需要调用np.expand_dims(img, 0)将输入shape扩为4维； 之后才可以传给rknn.inference()接口进行推理。通过cv2.imread()读取的图像的layout为NHWC，data_format的默认值也是NHWC，因此不需要设置data_format参数。
- 如果模型的输入数据不是通过cv2.imread()读取，此时必须清楚知道输入数据的layout并设置正确的data_format参数，同时也要确保输入数据的shape和原始模型一致。如果输入数据是图像数据，也要确保其RGB顺序与原始模型一致。

参数配置的检查是很重要的环节，是很多用户出现FP16推理输出结果错误的主要原因。具体排查步骤如下：

1. 使用原始模型在原始推理框架下进行推理，并将推理结果保存下来。
2. 使用RKNN-Toolkit2对原始模型进行转换并推理，此时需要使用与前一步骤里同样的输入数据，并设置FP16的推理方式（rknn.build() 的 do_quantization 设为 False），同时rknn.init_runtime()的target参数设为None，以调用RKNN-Toolkit2的模拟器进行推理，同样将推理的结果保存下来。
3. 对比两次推理的结果，如果结果较为一致（可以用余弦距离来判断一致性），说明上述的配置都没有问题。
4. 如果结果不一致，检查上述参数是否正确。

如果确认上述参数配置无误，结果仍然不一致，则有可能是模型中间的Tensor超出FP16表达范围导致。

- **超出FP16表达范围**

  模型的中间Tensor在由FP32转为FP16后，可能会出现运算溢出的问题。因为一般模型的推理数据类型是FP32，如果推理时模型中的Tensor有数值超过FP16表达范围（-65504～65504），则该Tensor就会溢出，导致模型推理结果异常。

  对于溢出问题，可以通过调用rknn.accuracy_analysis(..., target=None)接口（参考模型精度分析章节）进行模拟器FP16精度分析，如果分析结果中的simulator_error的entire列或single列出现异常值（出现'inf'等的字样），则可能出现了FP16溢出。此时可以尝试修改模型结构来保证模型中的所有Tensor不会出现FP16溢出（如添加一些BN层等）。

  如果确认上述参数配置无误，并且也不是FP16溢出，但结果仍然不一致，则可能是模拟器内部实现问题，请将该模型的复现文件提供给瑞芯微NPU团队进行分析解决。

7.1.2 模拟器量化精度

在排除FP16精度问题后，就可以对模型进行量化（使用rknn.build()接口时，将do_quantization参数设置为True），然后通过调用rknn.accuracy_analysis(..., target=None)接口（参考模型精度分析章节）进行模拟器量化精度分析。

如果在分析结果中，发现simulator_error的entire列精度下降的比较厉害，并且simulator_error的single没有发现有哪层精度下降特别多的情况，则主要从以下几个方面进行排查：

- **配置错误**

  与FP16推理的配置问题类似，错误的配置也会导致量化推理精度问题，因此在保证FP16推理正确的配置基础上，仍然要对以下量化配置参数进行检查。

  quant_img_RGB2BGR：表示在加载量化图像时是否需要先做RGB2BGR的操作，一般用于caffe模型，更多详细信息见 quant_img_RGB2BGR参数说明，该参数务必和训练时的图像通道顺序保持一致，在配置错误时也会导致量化精度下降比较多。

  optimization_level：优化等级的选择，默认为3，表示速度优先，这种情况下会开启一些对提升性能有益，但却会略微影响到精度的优化规则，将该配置调小（如改为0），则会禁用这些优化规则。

  dataset：rknn.build()接口的量化校正集配置，用于在量化过程中，计算每个Tensor合适的量化参数（scale / zero_point）。如果选择了和实际部署场景差异较大的校正集，则可能会出现精度下降的问题，此外校正集的数量过多或过少都会影响精度（一般选择20～200张）。

  具体检查量化参数配置问题，一般可按如下步骤进行：

  1. 直接进行量化推理，然后检查推理的结果与原始模型在原始推理框架下推理的结果进行比较，如果结果差异不是很大，则可以认为quant_img_RGB2BGR和dataset参数基本无误。
  2. 如果结果差异还是很明显：
     - 如原始模型输入的图像格式是BGR（多见于caffe的模型），此时可以修改quant_img_RGB2BGR为True，关于模型输入的RGB顺序，其实从前面FP16推理精度验证步骤的输入数据的处理代码中可以得知输入数据的RGB顺序。
     - 可以先使用一张图像进行量化（dataset.txt中只留一行），推理时也使用这张图像进行推理，如果此时单张图像的精度提升较多，则说明先前使用的量化校正集选择不佳，可以重新选择与部署场景较吻合的图片（如果提升并不明显，则可能不是dataset的问题）。
     - 如原先只使用一张图像进行量化（dataset.txt中只有一行），此时可以尝试使用更多的图像进行量化，可以提高到20～200张左右。

  经过上述配置排查之后，应该不会出现量化结果完全错误的情况，如果出现完全错误的情况，请重新检查上述的配置。在确认配置无误的情况下，如果模型的精度还是不够，可以尝试修改量化方法和算法等相关配置。

- **量化方法和量化算法**

  有些模型本身对量化并不友好，此时可以尝试切换不同的量化方法和量化算法。目前量化方法主要有两种，分别是layer和channel，可通过rknn.config()接口里的quantized_method参数进行设置（默认是channel）。量化算法主要分为三种，分别是normal， kl_divergence和mmse，可通过rknn.config()接口里的quantized_algorithm参数进行设置（默认是normal）。步骤如下：

  1. 如原先使用的是layer的量化方法，可以改为channel的量化方法，一般情况下，channel的量化方法精度比layer的量化方法精度会高许多。
  2. 如量化方法已经是channel，但精度还是无法满足要求，此时可以将量化算法由normal改为kl_divergence或mmse，这种方式会导致量化的时间大幅增加，但会带来比normal更好的精度表现，同时运行时的性能并不会受到影响。

  如果使用上述方法后，从分析结果中仍然发现simulator_error的entire列精度还是不好，并且simulator_error的single列有部分层精度掉的比较多，这可能是这些层的权重数值分布不好，导致量化后会出现精度下降较多的情况。如：Conv的weight的分布很不均匀的情况下，此时可以考虑使用**混合量化**来进一步提高模型精度。步骤如下：

  1. 先使用精度分析接口对精度进行分析，找出造成精度下降比较多的层，记录对应层的输出Tensor name。（这边需要注意的是，因为误差是会逐层累积的，所以越前面的层对最终的精度影响也会越大，因此不仅要考虑simulator_error的single列的精度损失情况，也要考虑层在模型中的位置）

2. 使用混合量化的方法，将上个步骤获取的Tensor name写入混合量化的配置文件中（参考混合量化章节）。

3. 完成混合量化的步骤，并测试精度情况（可以继续使用精度分析接口来看精度变化情况）。

一般经过混合量化之后，模型的精度是可以提高的，如果提高不明显或不够，则可以尝试将更多的层进行混合量化，但是同时也会造成推理性能下降，因此混合量化需要用户自行权衡精度和速度。还有一种特殊方式是，当精度下降的Op处于最后一层时，也可以选择将该层Op的运行放在后处理中进行，同样会有效避免该层的精度问题。

- **QAT量化感知训练**

  如果混合量化后精度还是不够，或者精度够但因混合量化而导致性能达不到要求，此时可以尝试使用量化感知训练重新训练模型，并导出带有量化参数的模型（如onnx/pt/pb/tflite格式）， 有关量化感知训练更多内容，请参考量化感知训练章节。

7.2 Runtime精度排查

在模拟器精度正常的情况下，仍可能在板端C API部署时出现推理结果异常。出现这种问题的原因一般有三种，第一种是板端的Runtime的bug导致；第二种是调用RKNPU2的C API编程时接口没有正确使用导致；第三种是模型的前后处理不正确导致。

当遇到这种问题时，可以先通过连板功能快速排查是否是板端Runtime的bug导致，如果连板没有问题，再排查C API部署的问题。

7.2.1 连板精度

1. 在配置好连板调试环境的情况下（连板调试环境配置方法参考设备端NPU环境准备章节），将开发板通过USB连接到电脑上，然后使用RKNN-Toolkit2进行连板推理（设置rknn.init_runtime()的target参数，如target='rk3566'），并检查推理结果是否大致正确（因为模拟器并没有严格模拟NPU硬件，所以结果可能与模拟器并没有完全一致）。
2. 如果上述步骤里的推理结果与模拟器推理结果差异较大，则可以初步确定板端的Runtime存在bug，此时可以使用精度分析的接口（参考模型精度分析章节）进行连板精度分析（调用rknn.accuracy_analysis()接口，并设置target参数即可，如target='rk3566'），精度分析完后会输出每层的分析结果。
3. 检查分析结果中的runtime_error的single_sim列，如其cos余弦距离偏低或euc欧氏距离偏高（显示黄色或红色），从而导致runtime_error的entire列与simulator_error的entire列差异越来越大，则可能Runtime在实现该层时有出现精度丢失或异常的问题，此时可以将该分析结果以及复现的模型反馈给瑞芯微NPU团队进行修复。

7.2.2 Runtime精度

如果连板精度没有问题，但精度仍然有问题，则问题可能出在用户调用RKNN的C API进行编程的C/C++代码本身，这时用户需要仔细检验下RKNN的C API的接口配置等是否配置正确，以及模型的前处理和后处理流程是否正确（需要与模拟器端的流程完全一致）。可以按照以下步骤查看：

1. **检查输入配置和数据**

   查看C API的输入是否配置正确。例如，RKNN-Toolkit2在转换RKNN模型时已经配置均值和方差，则在C/C++代码中不需要做归一化。对于3通道的输入，通道顺序与模型训练时设置的输入通道一致；对于四维输入形状，fmt=NHWC；对于非四维输入，fmt=UNDEFINED。若使用通用API，输入buffer的size等于输入Tensor元素个数*每个元素的字节数，若使用零拷贝API，rknn_create_mem接口创建的内存大小以及输入数据格式参考《RKNN Runtime零拷贝调用》章节。

在确认配置正确后，需查看输入层的数据，可以在应用运行前设置RKNN_LOG_LEVEL=5，然后再运行应用，推理时逐层的结果会以numpy格式文件保存在/data/dumps（Android系统）或/userdata/dumps（Linux系统）目录下。查看包含InputOperator字段的numpy文件是否符合预期，如果使用通用API，它是输入归一化的结果；如果使用零拷贝API，它是未归一化前的数据。

2. **检查输出配置和数据**

   在确保输入正确后，查看代码中输出是否配置正确。例如，如果使用通用API，当设置want_float=1后，输出是float32类型结果，当设置want_float=0后，输出是量化数据类型或者float16类型(非量化数据类型)。如果使用零拷贝API，rknn_create_mem()接口创建的内存大小以及输入数据格式参考《RKNN Runtime零拷贝调用》章节。

   查看输出层的数据，同样在上述逐层numpy文件生成后，打开包含OutputOperator字段的numpy文件，查看数据是否正确。如果确认输入结果正确，输出仍然错误，可能Runtime在特定的输入/输出类型处理上有问题，此时可以将该分析结果以及复现的模型反馈给瑞芯微NPU团队进行修复。

8 性能优化

模型部署时，用户在跑通结果后，会有进一步的性能优化需求，此节将从完整的模型性能优化流程来介绍如何调优。并展开介绍用户最常用的操作：模型性能分析，图级别优化，算子级别优化。完整优化流程如下图所示：

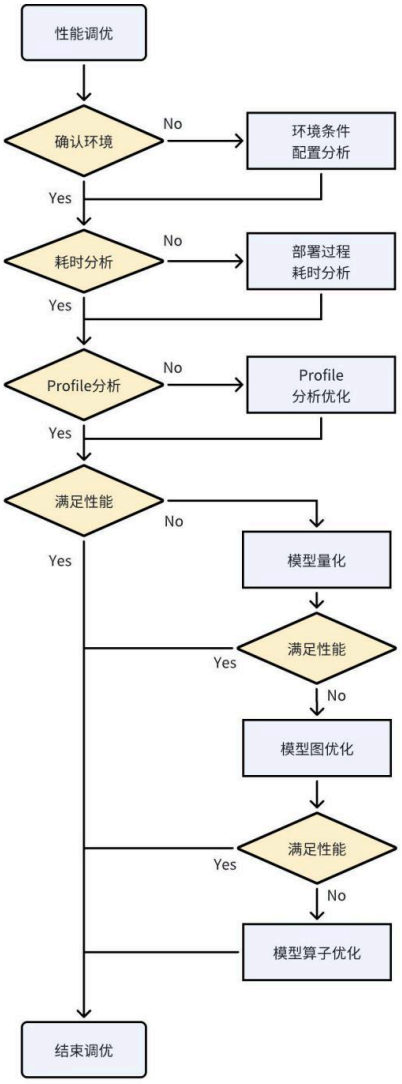


图8-1 模型性能优化流程

8.1 模型性能优化前期分析流程

8.1.1 环境条件与配置检查

在所有性能分析及优化之初，应该优先确定测试环境的基准，只有在基准相同时，所测性能数据才是有意义的。例如非定频下测试同一模型的推理性能波动幅度是比较大的，无法以某一批测试耗时作为平均的单帧耗时。

查询和设置测试环境的条件和配置有如下几个方面：

- **查询和设置CPU、DDR、NPU频率**

  定频频率直接影响运行速率，频率越高性能相对越好。频率变化与性能提升不是线性关系。频率增加相对来说功耗也会增加。定频命令参考如下：https://github.com/airockchip/rknn_model_zoo/blob/main/scaling_frequency.sh

  也可以简单的使用如下命令设置为性能模式：

```
echo performance | tee $(find /sys/ -name *governor) /dev/null || true
```

- **检查NPU内核Driver版本**

  有些功能或者算子的性能优化项与内核驱动版本有关，较新的内核驱动版本能应用到最新的底层优化实现。所以请检查是否使用到较新的内核驱动版本，目前建议更新到0.9.2之后的版本。该环节非强制更新。

  检查内核驱动版本命令如下：

```
cat /sys/kernel/debug/rknpu/version     # for RK3566/RK3568/RK3588/RK3562/RK3576
cat /proc/rknpu/version                         # for RV1106系列/RV1103系列/RV1126B
```

- **检查NPU的负载**

  NPU的负载为单位时间内NPU执行任务的时间占比。负载能够反应NPU的繁忙程度，如果查询到的负载较低，则表明NPU等待任务提交的时间较长，需要检查数据输入输出拷贝用时、应用程序前后处理优化等等。或者在应用程序中使用多线程处理方式来提升NPU负载。

  另外需要注意的是，NPU的负载不代表实际的MAC利用率，MAC利用率反应的是执行中任务在NPU硬件单元中的执行效率。

  查询NPU负载的命令如下：

```
cat /sys/kernel/debug/rknpu/load
# or
cat /proc/debug/rknpu/load
```

8.1.2 部署过程耗时分析

整个部署程序的推理部分耗时的占用有如下三个方面：用户应用程序耗时、输入输出数据拷贝耗时、模型推理耗时。分析各环节耗时占比能够直观的确定优化重点。测定这些步骤的耗时可以在应用程序上通过打时间戳的方式获得。

- **用户应用程序耗时**

  用户应用程序耗时主要指推理过程中非NPU相关的耗时，一般来说主要是数据的前处理、后处理和逻辑代码的耗时。这部分耗时由用户全权控制。用户在发现应用程序的耗时占总体耗时非常高时，除精简优化代码以外，可以尝试将一部分操作通过专用硬件加速。

  例如将一些矩阵乘加操作，采用Matmul API接口来调用NPU辅助执行计算。又如部分图像的缩放类操作可以调用RGA的接口来实现加速。

- **输入输出拷贝耗时**

  当采用通用API时，用户设置的输入输出内存与NPU的内存是存在拷贝耗时的，这个耗时可以在调用通用API时打印出来。拷贝耗时取决于DDR与CPU的性能，在输入输出数据量较小的时候Normal API的拷贝耗时较低，但数据量较大时，Normal API的耗时就不可忽略。因此更多推荐采用零拷贝API。

  当采用零拷贝API时，用户设置的输入输出内存被NPU直接访问，所以输入输出拷贝耗时为0。零拷贝的接口的使用详细参考RKNN Runtime零拷贝调用章节。

- **推理耗时**

  NPU执行推理的耗时，该部分耗时直接体现部署模型的耗时。受推理模型规模、编译优化版本影响。RKNN的LOG打印的推理耗时在不同 RKNN_LOG_LEVEL等级时不一样，因为LOG打印存在一定的耗时。一般在查看单帧推理耗时时，设置的LOG等级为1，并且跑多次取平均为准。

8.2 模型性能分析

8.2.1 获取Profile信息

当需要了解模型推理逐层耗时情况时，可以在运行程序前输入以下指令打印详细信息：

```
export RKNN_LOG_LEVEL=4
./run_rknn_test ./test.rknn ./input.jpg
```

如果是Android平台，运行后可以通过logcat命令获取详细日志。

如果是使用rknn-toolkit2，你可以使用如下方式来获取每层的耗时：

```
rknn.init_runtime(target=platform, perf_debug=True)
rknn.eval_perf()
```

性能分析报告信息如下（仅截出性能相关部分）

```
-------------------------------------------------------------------------------------------------
                                Operator Time Consuming Ranking Table
-------------------------------------------------------------------------------------------------
OpType            CallNumber    CPUTime(us)  GPUTime(us)  NPUTime(us)  TotalTime(us)    TimeRatio(%)
ConvLeakyRelu     7             0            0            4584         4584             53.77%
MaxPool           6             0            0            2273         2273             26.66%
Conv              2             0            0            846          846              9.92%
ConvAdd           1             0            0            511          511              5.99%
Split             1             0            0            152          152              1.78%
LeakyRelu         1             0            0            68           68               0.80%
OutputOperator    1             62           0            0            62               0.73%
InputOperator     1             29           0            0            29               0.34%
-------------------------------------------------------------------------------------------------
```

图8-2 性能分析报告

```
ID   OpType          DataType Target InputShape                                  OutputShape          DDRCycles    NPUCycles    MaxCycles    Time(us)
-------------------------------------------------------------------------------------------------------------------------------------------------
0    InputOperator   UINT8    CPU    \                                           (1,3,480,640)        0            0            0            9
1    ConvExSwish     UINT8    NPU    (1,3,480,640),(32,3,3,3),(32)               (1,32,240,320)       794117       1382400      1382400      2805
2    ConvExSwish     INT8     NPU    (1,32,240,320),(64,32,3,3),(64)             (1,64,120,160)       665207       691200       691200       1605
3    ConvExSwish     INT8     NPU    (1,64,120,160),(32,64,1,1),(32)             (1,32,120,160)       332036       153600       332036       832
4    ConvExSwish     INT8     NPU    (1,64,120,160),(32,64,1,1),(32)             (1,32,120,160)       332036       153600       332036       788
5    ConvExSwish     INT8     NPU    (1,32,120,160),(32,32,1,1),(32)             (1,32,120,160)       248988       153600       248988       722
6    ConvExSwish     INT8     NPU    (1,32,120,160),(32,32,3,3),(32)             (1,32,120,160)       250087       345600       345600       768
7    Add             INT8     NPU    (1,32,120,160),(1,32,120,160)               (1,32,120,160)       329574       0            329574       298
8    Concat          INT8     NPU    (1,32,120,160),(1,32,120,160)               (1,64,120,160)       494407       0            494407       453
9    ConvExSwish     INT8     NPU    (1,64,120,160),(64,64,1,1),(64)             (1,64,120,160)       497258       307200       497258       1427
10   ConvExSwish     INT8     NPU    (1,64,120,160),(128,64,3,3),(128)           (1,128,60,80)        401854       691200       691200       962
11   ConvExSwish     INT8     NPU    (1,128,60,80),(64,128,1,1),(64)             (1,64,60,80)         167682       76800        167682       405
```

图8-3 性能分析报告

可以针对总体性能Profile和逐层性能Profile快速定位想要的信息。并根据数据来制定后续不同侧重的优化策略。获得Profile后可以进行如下分析：分析逐层耗时，找出高耗时算子；分析非NPU算子影响；分析NPU算子性能瓶颈。下文将详细讨论。

8.2.2 分析逐层耗时

如下图中，可以从Time一栏找出耗时高的算子，优先优化高耗时算子。也可以兼顾观察Op Type一栏找出高耗时的算子是不是属于同类OpType，以便统一优化。

ID	OpType	DataType	Target	InputShape	OutputShape	DDRCycles	NPUCycles	MaxCycles	Time(us)
117	Sigmoid	INT8	NPU	(1,83,60,80)	(1,83,60,80)	186625	0	186625	541
118	ConvRelu	INT8	NPU	(1,128,60,80),(128,128,3,3),(128)	(1,128,60,80)	327096	1382400	1382400	1595
119	ConvRelu	INT8	NPU	(1,128,60,80),(128,128,3,3),(128)	(1,128,60,80)	327096	1382400	1382400	1596
120	Conv	INT8	NPU	(1,128,60,80),(4,128,1,1),(4)	(1,4,60,80)	103405	19200	103405	150
121	Conv	INT8	NPU	(1,128,60,80),(1,128,1,1),(1)	(1,1,60,80)	103345	19200	103345	150
122	Sigmoid	INT8	NPU	(1,1,60,80)	(1,1,60,80)	32181	0	32181	142
...									
144	Mul	INT8	NPU	(1,83,60,80),(1,1,60,80)	(1,83,60,80)	319655	0	319655	414
145	ReduceMax	INT8	CPU	(1,83,60,80)	(1,1,60,80)	0	0	0	7441
146	ArgMax	INT8	CPU	(1,83,60,80)	(1,1,60,80)	0	0	0	7389
147	Reshape	INT64	CPU	(1,1,60,80),(3)	(1,60,80)	0	0	0	69
148	OutputOperator	INT64	CPU	(1,60,80)	\	0	0	0	13

图8-4 高耗时算子性能分析

值得说明的是，高耗时算子不一定是低效算子，某些算子是高算力消耗的，高耗时的算子如果Mac利用率很高时，应该考虑能否降低此算子的尺寸规模以减少耗时。但当利用率很低的高耗时算子出现时，这类算子就是优化重点。

8.2.3 分析CPU算子影响

如下图中，可以看到某些高耗时的算子是运行在CPU上的，将这些CPU算子NPU化将可以极大改善高耗时影响。一般来说，用户的大部分性能优化问题都会在将CPU算子NPU化后得到解决。因此要重点注意CPU算子的耗时情况。

Operator Time Consuming Ranking Table

OpType	CallNumber	CPUTime(us)	GPUTime(us)	NPUTime(us)	TotalTime(us)	TimeRatio(%)
ConvRelu	125	0	0	54613	54613	42.09%
ConvExSwish	35	0	0	20047	20047	15.45%
ReduceMax	7	16104	0	0	16104	12.41%
ArgMax	7	14833	0	0	14833	11.43%
Concat	36	0	0	10464	10464	8.06%
Conv	25	0	0	3685	3685	2.84%
exSoftmax13	1	0	0	1916	1916	1.48%
Resize	11	0	0	1893	1893	1.46%
Sigmoid	12	0	0	1595	1595	1.23%
Reshape	14	388	0	699	1087	0.84%
Mul	7	0	0	1065	1065	0.82%
Add	7	0	0	1046	1046	0.81%
MaxPool	9	0	0	784	784	0.60%
OutputOperator	32	569	0	0	569	0.44%
ConvSigmoid	1	0	0	40	40	0.03%
InputOperator	1	9	0	0	9	0.01%

图8-5 CPU算子性能分析

一般来说算子运行在非NPU上的原因有如下几种：

- 算子尺寸超限（查询OpList文档的算子尺寸限制）
- 算子尚未支持在NPU上运算（查询OpList是否支持该算子，可以在Github工程上提Issue）
- NPU硬件限制无法支持（是否可以算法等效成其他NPU可支持的其他实现）

8.2.4 分析NPU算子性能瓶颈

考虑NPU算子的高耗时问题时，可以根据DDR Cycles/NPU Cycles/Total Cycles这三栏来判断该算子耗时的理论瓶颈是带宽瓶颈还是算力瓶颈。这里的DDR Cycles是根据NPU频率换算过后的数据，指该层算子读写数据换算成NPU频率下所需的Cycle数，因此可以直接与NPU Cycle比较。

如下图所示：

ID	OpType	DataType	Target	InputShape	OutputShape	DDR Cycles	NPU Cycles	Total Cycles	Time(us)	MacUsage(%	Task Number	Regcmd Size	RV
0	InputOperatc	UINT8	CPU	\	(1,3,300,300)	0	0	0	12	\	0	0	
1	Conv	UINT8	NPU	(1,3,300,300)	(1,3,300,300)	261543	1582	261543	585	0.3	0	0	1
2	Split	INT8	CPU	(1,3,300,300)	(1,1,300,300),(1	0	0	0	6677	\	0	0	
3	Conv	INT8	NPU	(1,1,300,300)	(1,8,150,150)	275553	17226	275553	1556	1.23	0	0	1
4	Conv	INT8	NPU	(1,1,300,300)	(1,8,150,150)	275553	17226	275553	1505	1.27	0	0	1
5	Conv	INT8	NPU	(1,1,300,300)	(1,8,150,150)	275553	17226	275553	1573	1.22	0	0	1
6	Concat	INT8	CPU	(1,8,150,150)	(1,24,150,150)	0	0	0	4684	\	0	0	1
7	Conv	INT8	NPU	(1,24,150,15(	(1,64,150,150)	329723	67500	329723	684	10.96	0	0	2
8	Clip	INT8	NPU	(1,64,150,15(	(1,64,150,150)	438387	0	438387	729	\	0	0	2
9	MaxPool	INT8	NPU	(1,64,150,15(	(1,64,75,75)	275490	0	275490	725	\	0	0	1
10	Conv	INT8	NPU	(1,64,75,75),(	(1,64,75,75)	110676	45000	110676	281	17.79	0	0	
11	Clip	INT8	NPU	(1,64,75,75),(	(1,64,75,75)	109676	0	109676	275	\	0	0	
12	Conv	INT8	NPU	(1,64,75,75),(	(1,192,75,75)	253595	1215000	1215000	1456	92.72	0	0	1
13	Clip	INT8	NPU	(1,192,75,75)	(1,192,75,75)	328817	0	328817	599	\	0	0	2
14	MaxPool	INT8	NPU	(1,192,75,75)	(1,192,38,38)	206599	0	206599	535	\	0	0	1
15	AveragePool	INT8	CPU	(1,192,38,38)	(1,192,38,38)	0	0	0	16159	\	0	0	
16	Conv	INT8	NPU	(1,192,38,38)	(1,32,38,38)	50564	17328	50564	192	10.03	0	0	
17	Clip	INT8	NPU	(1,32,38,38),(	(1,32,38,38)	14169	0	14169	121	\	0	0	
18	Conv	INT8	NPU	(1,192,38,38)	(1,64,38,38)	58609	34656	58609	193	19.95	0	0	
19	Clip	INT8	NPU	(1,64,38,38),(	(1,64,38,38)	28233	0	28233	132	\	0	0	
20	Conv	INT8	NPU	(1,64,38,38),(	(1,96,38,38)	43855	155952	155952	304	57	0	0	
21	Clip	INT8	NPU	(1,96,38,38),(	(1,96,38,38)	42297	0	42297	152	\	0	0	
22	Conv	INT8	NPU	(1,96,38,38),(	(1,96,38,38)	55095	233928	233928	399	65.14	0	0	
23	Clip	INT8	NPU	(1,96,38,38),(	(1,96,38,38)	42297	0	42297	153	\	0	0	
24	Conv	INT8	NPU	(1,192,38,38)	(1,64,38,38)	58609	34656	58609	144	26.74	0	0	

图8-6 NPU算子性能瓶颈分析

- 其中第三层 DDR Cycles 远大于 NPU Cycles时，说明该层读写数据花费Cycle数量远大于运算所需Cycle数量，所以该Conv瓶颈来自带宽。
- 其中第十二层 DDR Cycles 远小于 NPU Cycles时，说明该层读写数据花费Cycle数量远小于运算所需Cycle数量，所以该Conv瓶颈来自算力。

目前NPU Cycles一栏主要显示Conv所需的Cycles，其他算子类型后续会逐步补充。

8.3 量化加速

模型量化能大幅降低模型的运算规模，节约带宽消耗。由于采用了INT8的量化的卷积采用的是INT8的运算单元计算。同时算力上，相比于Float16的运算单元，INT8的运算单元规模更大，理论算力更高。模型量化的具体使用方式详见量化说明。

8.4 图级别优化

模型的图级别优化是最容易从整体角度去统筹优化模型的方法。在分析出耗时占比较高的算子或图区域后，我们可以有多种不同的方式去改造图进而达成优化的目的。图优化主要以节省多余算子、非NPU OP的NPU化、面向硬件高效率算子改造等为目标。这些目标有可能有些时候是存在矛盾的，例如为了非NPU OP的NPU化，可能需要额外多出几个算子，看似违背了节省多余算子的目标，但总体推理性能提升，便是有意义的。

在RKNN-Toolkit2工具链中，软件栈在转换模型的过程中已经会进行一定程度上的图优化。但这一过程不是万能和尽善尽美的，有些未被考虑的场景仍然会出现冗余的操作，用户可以根据本节介绍的一些思路来进行预先性的图优化。以下仅作为每一种优化方法的介绍，不是强制固定，实际场景需要灵活运用。

8.4.1 非NPU OP通过图变换实现NPU化

对于非NPU op，可以做一些等效的图变换来，替换成NPU可支持的算子，以达成NPU化的优化目的。

例如下图，以shufflenetv2_0.5模型为例，将其中channel shuffle操作改为卷积近似替换。weight数值为0/1，可以达成重排数据的效果，假如无法在NPU实现该Transpose、Reshape操作，可以将这些算子整合成Conv算子实现数据重排。

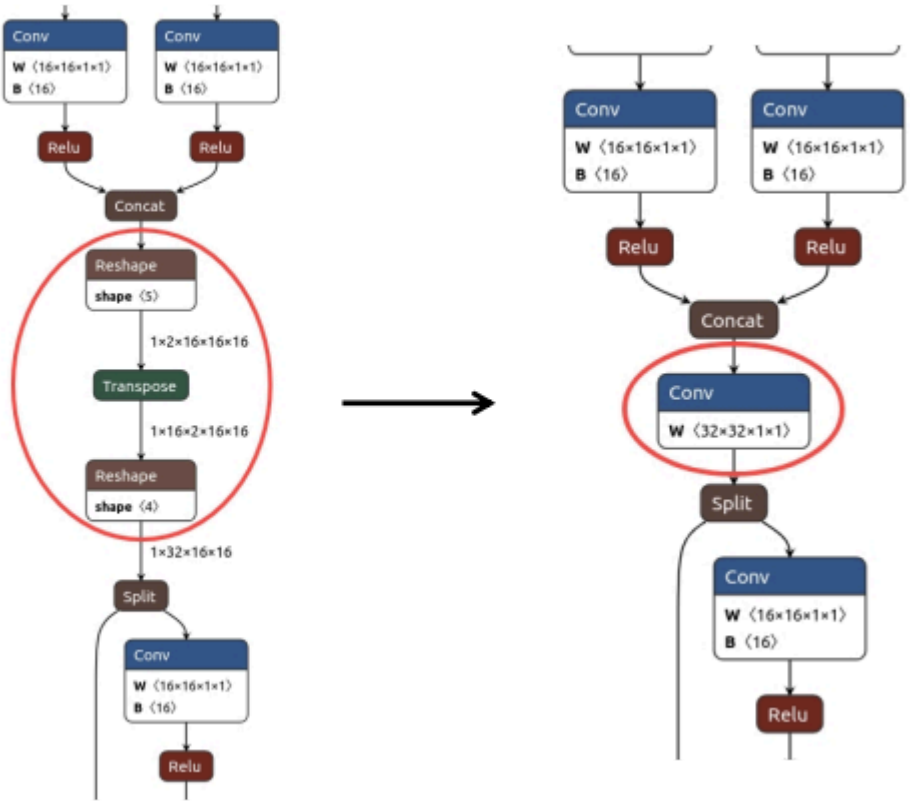


图8-7 卷积重排

8.4.2 利用硬件Fuse特性设计网络或图优化

NPU支持一些算子组合进行融合，可以适当调整部分算子的运算流程，以适应NPU的融合规则实现算子融合优化。

尽管RKNN软件栈会有一定程度的图优化，但无法做到全面覆盖到所有情况。某些特殊情况下出现了理论上可融合简化，但最终图优化未能融合的图结构，用户可以算法上手动调整以快速解决该优化问题。

例如下图，在不改变计算正确性的情况下，通过调整Transpose与Clip算子的顺序，使得Conv与Clip融合运算，提高了性能。

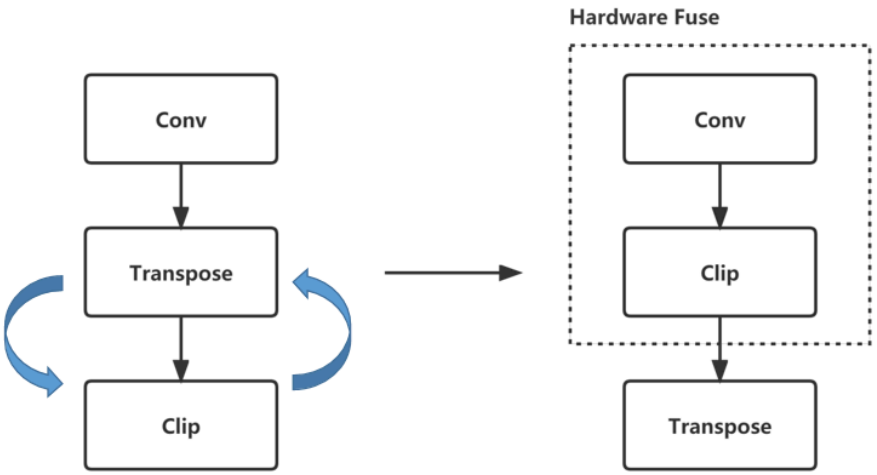


图8-8 算子图优化/融合

利用融合规则设计，融合规则如下：

已支持的融合规则	未来计划支持的融合规则
Conv+Relu	Activation+Add(Mul)
Conv+PRelu(LeakyRelu)	Add(Mul)+Activation
Conv+Clip	Conv+Mul
Conv+Sigmoid(Tanh/Elu/Silu...)	Conv+Activation+Mul
Conv+Add	Conv+Activation+Pooling
Conv+Activation+Add	Conv+Activation+Add(Mul)+Pooling

8.4.3 算法等效变换或者子图单OP化

在分析某个图区域时，有时算法上要实现某个行为功能可能设计上追求表达的直白性不会考究具体部署上的性能影响，容易产生出复杂冗余的图结构，可以通过算法等效的方式，将某个区域的子图单op化，减少算子计算步骤，达成优化目的。

例如下图为Yolov5-nano 等效图变换，将若干复杂的Slice取数融合到Conv中，形成一个新的Conv，极大简化了图结构。

方案来源：https://github.com/ultralytics/yolov5/issues/4825

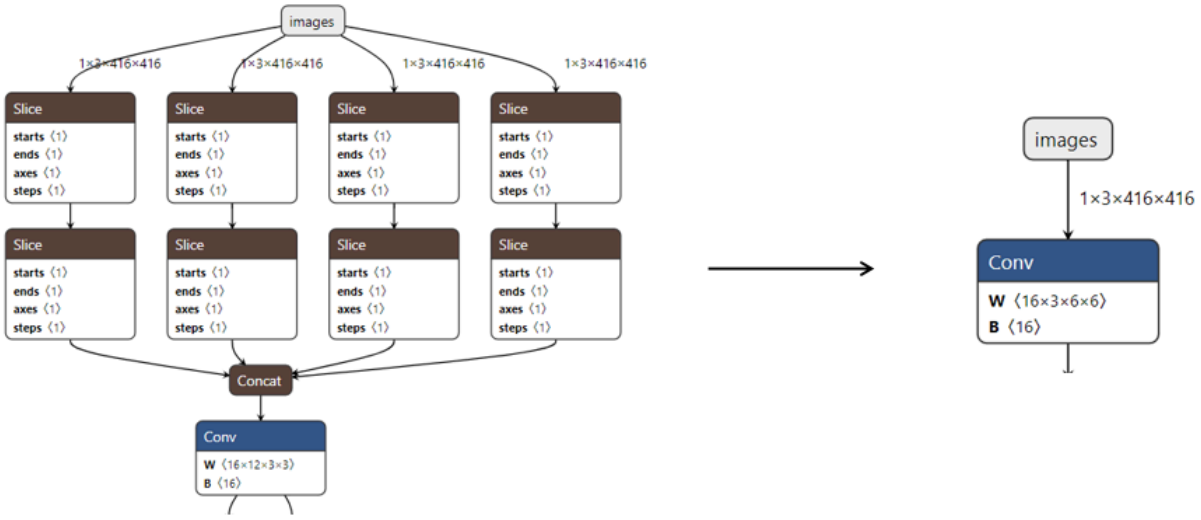


图8-9 算子等效图变换

8.4.4 算子等效进行“同类项合并”、“提取公因式”

某些算子连续多次运算时，可以简省合并为同一个算子，如Reshape、Transpose、Slice、部分Add/Mul/Sub/Div等。

例如下图可以通过简单调整图顺序以达成同类算子合并目的。

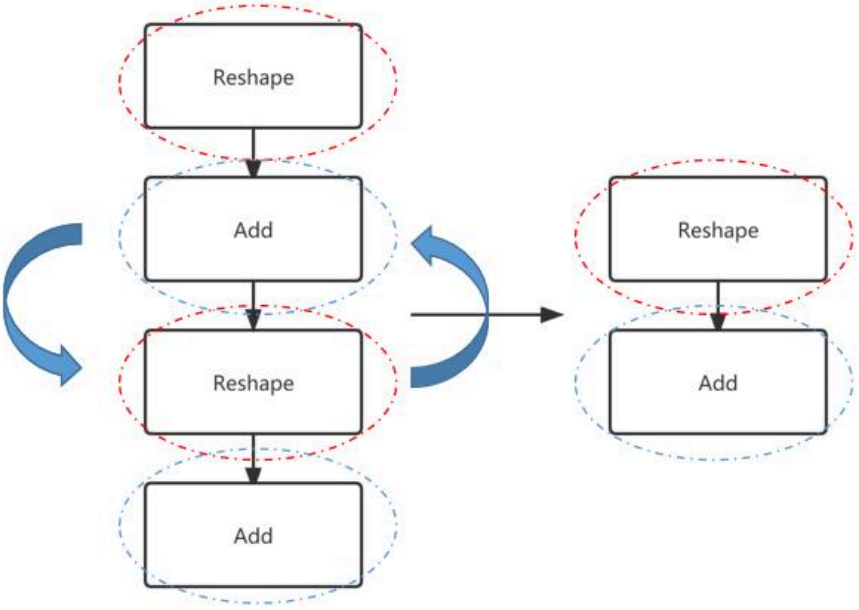


图8-10 同类项算子合并

某些图结构有一些共有部分的同类型操作可以调整顺序以提取成单一操作。

例如下图可以通过调整算子顺序将重复性的同类算子单独提取出来只执行一次操作。

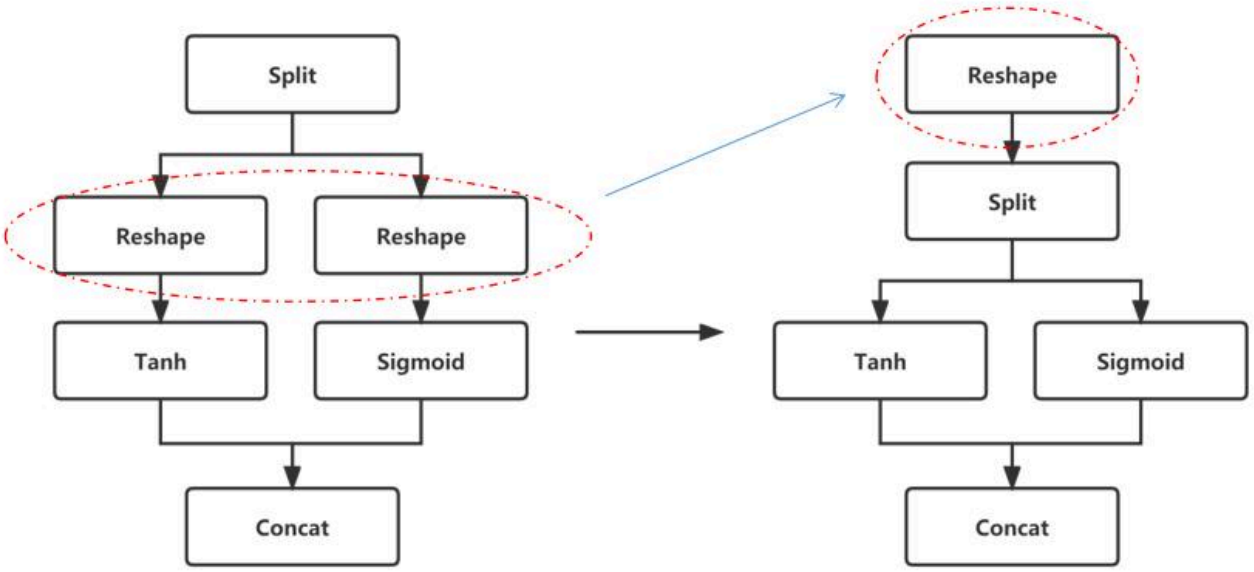


图8-11 重复性算子合并

8.5 算子级别优化

模型的算子级别优化是针对性比较强的细节优化，对于某些特定算子的具体改造设计，以期进一步提升性能。算子的优化更多是针对性进行算子尺寸设计，以硬件实现效率最高的尺寸规格运行，例如某些算子尺寸规模相似，对齐与非对齐的运行耗时可能差别巨大，差别的原因在于硬件对于部分非对齐尺寸的算子会需要额外的冗余操作来保证正确性，因此算子的尺寸设计对于模型性能也能起到很大的影响，用户可以根据如下一些思路来进行预先性的算子优化。

8.5.1 面向DDR性能优化的OP尺寸设计（非强制）

在一些对齐尺寸下，除NPU运算效率更高外，对于DDR的读写也更友好，同等带宽条件下，更友好的读写会提升DDR的带宽效率，从而达到更好的性能。以下列出一些对于DDR读写更友好的尺寸规则，这些规则不强制。

- **Channel按对齐量对齐**

  对齐表格如下所示：

表8-1 RK3566/RK3568

	Conv		**Depthwise Conv**	**Other OP**
Dtype	InputChannel	OutputChannel	Inputl & Output Channel	Inputl & Output Channel
Int8	32	16	32	8
Int16	16	8	16	4
Float16	16	8	16	4
BFloat16	16	8	16	4

表8-2 RK3588/RK3576/RV1126B

	Conv		Depthwise Conv	Other OP
Dtype	InputChannel	OutputChannel	Inputl & Output Channel	Inputl & Output Channel
Int8	32	32	64	16
Int16	32	16	32	8
Float16	32	16	32	8
BFloat16	32	16	32	8
TFloat32	16	16	16	4

表8-3 RV1106/RV1103

	Conv		Depthwise Conv	Other OP
Dtype	InputChannel	OutputChannel	Inputl & Output Channel	Inputl & Output Channel
Int8	32	16	32	16
Int16	16	16	16	8

表8-4 RK3562

	Conv		Depthwise Conv	Other OP
Dtype	InputChannel	OutputChannel	Inputl & Output Channel	Inputl & Output Channel
Int8	32	16	32	16
Int16	32	8	16	8
Float16	32	8	16	8
BFloat16	32	8	16	8
TFloat32	16	8	8	4

- Height * Width> 1时4对齐
- 同等规模的算子，Width大Height小的尺寸，面向DDR读写更友好。如下图所示，右图卷积效率高于左图卷积

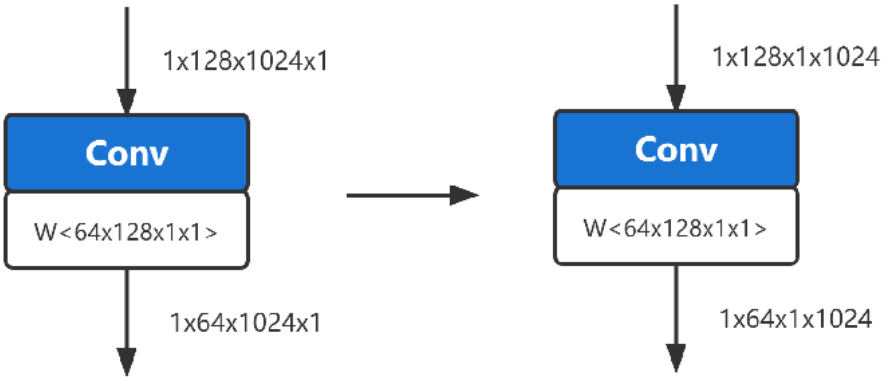


图8-12 同等规格卷积对比

8.5.2 高利用率模型算子的设计

要在模型设计上整体提高算力的利用率，一般要尽量避免低效算子，以及选择容易跑出更高利用的算子尺寸设计。这里主要列出一些可以尽可能避免的低效算子、讨论卷积尺寸与利用率的关系。

- **规避低效算子原则设计**

  模型中尽量减少低效算子的使用，低效算子主要有三类，如表8-5所示

  表8-5 三类低效算子

数据搬运类	尺寸变换类	非Relu类激活函数
Transpose	Resize	Sigmoid
Reshape	Tile	Tanh
Split	Pooling	Softplus
Concat	Pad	Hardswish

- **卷积尺寸与利用率关系的讨论**

  由于卷积的性能会受到算力和带宽的双重影响，在评估性能时常采用MAC利用率来说明硬件算力发挥程度。

  卷积尺寸与硬件算力和带宽读写相关，因此这里讨论卷积尺寸与利用率关系，作为用户设计模型的性能参考辅助。以下根据经验数据来作为一个大致的参考：

  以下名词注释：KH（kernel height），KW（kernel width），KC（kernel channel），type_bytes（权重位宽除以8），Ksize（KW或KH），Kstride（KW或KH方向上的stride）

  - 卷积的输入输出Tensor的Channel符合对齐要求时（见8.4.1中对齐表格数据），利用率更高。
  - 输入Tensor的 Channel < 256 时利用率相对较高，当Channel > 512以后，随着Channel增大，利用率会逐渐下降。
  - 权重尺寸上，KH* KW * KC * type_bytes < 6K Bytes 时利用率相对较高。当超过一定大小后利用率会明显下降。
  - Ksize / Kstride 的比值越大，利用率相对更高。例如(Ksize=3, Kstride=1优于Ksize=2, Kstride=1)
  - 输出Tensor的Height * Width < 16 时利用率下降。
  - 输出Tensor的Channel越大，利用率越高。

以上讨论仅是考察独立尺寸影响利用率的因素，实际部署模型里的卷积所呈现出的实测利用率则是诸多因素综合后的结果，开发者如果对某一卷积性能不够满意，希望通过提升利用率以优化其性能，可以参考上述尺寸与利用率关系的讨论进行针对性调整。

8.5.3 子图融合的匹配

RKNN软件栈会将某些特定的图关系匹配成自定义算子，如下图所示，如果没有被融合成对应的算子，可以考察一下是否连接关系不同没有匹配上。

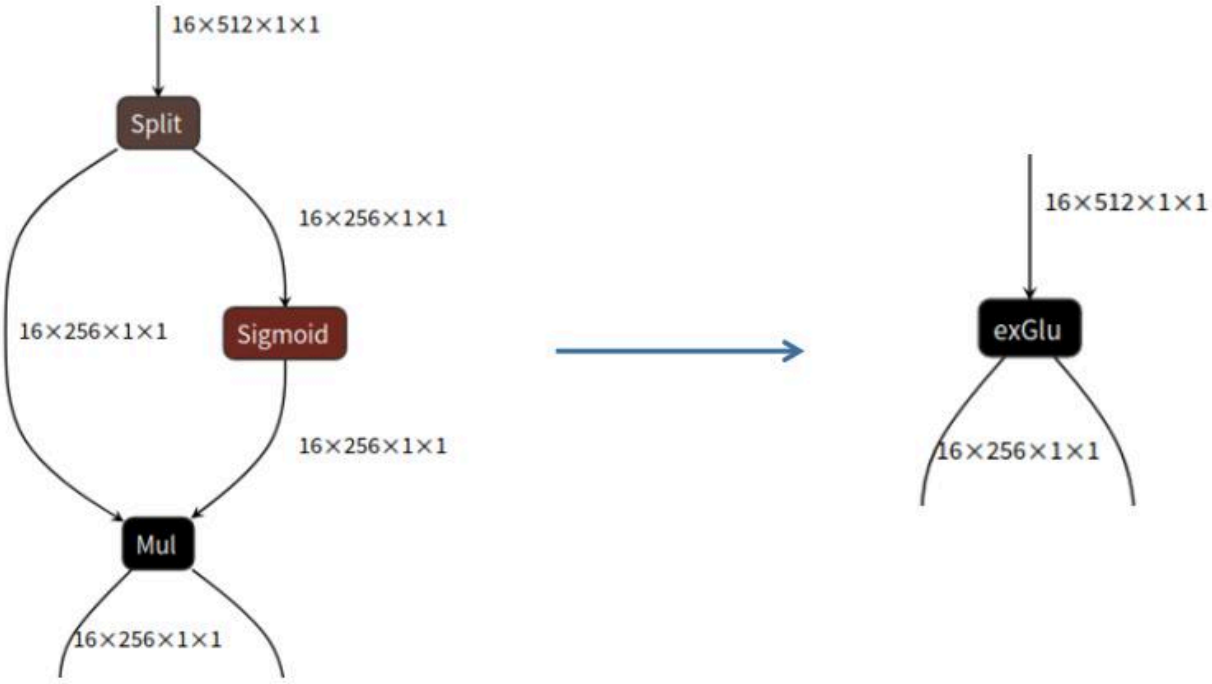


图8-13 Glu子图融合

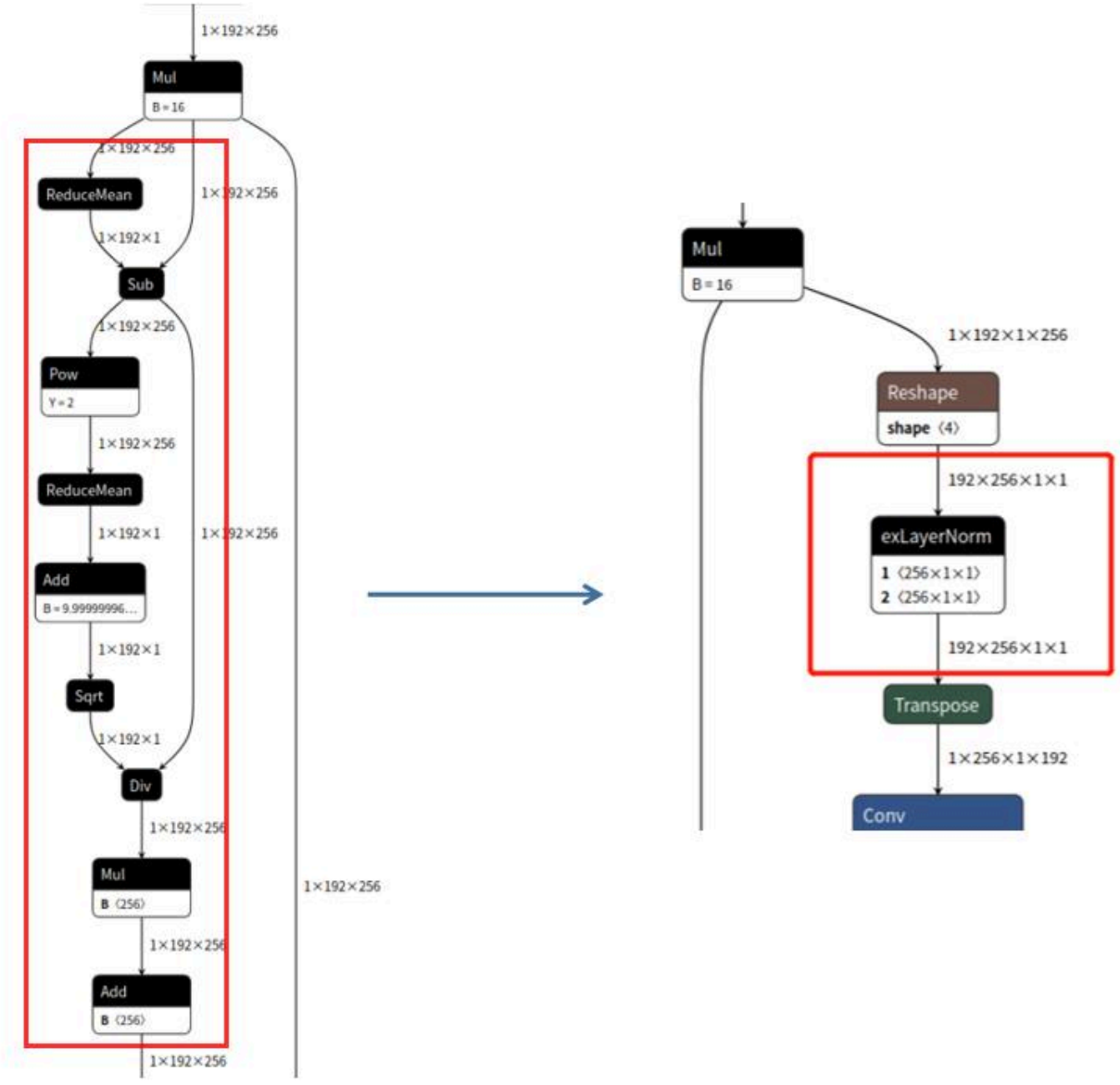


图8-14 LayerNorm子图融合

目前已经支持的子图融合规则有：

- Split + Sigmoid + Mul -> GLU
- ReduceMean + Sub + Pow + ReduceMean + Add + Sqrt + Div (+ Mul + Add) -> LayerNorm

9 内存使用优化

9.1 模型运行时内存组成及分析方法介绍

9.1.1 RKNN模型运行时内存组成

RKNN 模型运行时内存主要由权重和internal tensor、寄存器配置、输入输出tensor四种组成。运行时的内存通常是在rknn_init的时候创建完成。

9.1.2 模型内存分析方法

在rknn_init()接口调用完毕后，当用户需要查看模型分配的内存或者需要外部分配模型权重的时候，调用rknn_query接口，传入RKNN_QUERY_MEM_SIZE即可查询模型的权重、internal的内存(不包括输入和输出)、模型推理所用的所有 DMA 内存以及 SRAM 内存（如果SRAM没开或者没有此项功能则为0）的占用情况。

以下是示例代码：

```
rknn_context ctx = 0;

// Load RKNN Model
int ret = rknn_init(&ctx, model_path, 0, NULL, NULL);
if (ret < 0) {
    printf("rknn_init fail! ret=%d\n", ret);
    return -1;
}

// Get weight and internal mem size
rknn_mem_size mem_size;
ret = rknn_query(ctx, RKNN_QUERY_MEM_SIZE, &mem_size, sizeof(mem_size));
if (ret != RKNN_SUCC) {
    printf("rknn_query fail! ret=%d\n", ret);
    return -1;
}
printf("total weight size: %d, total internal size: %d\n", mem_size.total_weight_size,
mem_size.total_internal_size);
```

9.2 如何使用外部分配内存

9.2.1 输入输出内存外部分配

根据章节C API零拷贝整体流程里提到的，如果用户使用零拷贝API，可以在外部分配内存给输入输出tensor，然后配置给NPU使用，具体流程可以参照C API零拷贝整体流程里的流程图。注意，要用外部内存分配的方式，只能使用零拷贝API。该方法主要适用场景是用户需要手动分配内存给NPU使用，而不是通过rknn_create_mem()接口来让NPU自己分配内存。

外部内存可以用物理地址和fd记录，主要通过下面2个接口创建：

- rknn_create_mem_from_phys()：通过物理地址来创建rknn_tensor_mem的结构体
- rknn_create_mem_from_fd()：通过fd来创建rknn_tensor_mem的结构体

这里提供了一个用mpi mmz创建内存的例子。该例子通过rknn_create_mem_from_phys()接口，引入外部内存的物理地址， 创建一个rknn tensor mem的结构体带物理内存的信息。以下是示例代码：

```
// Create input tensor memory
rknn_tensor_mem* input_mems[1];
// default input type is int8 (normalize and quantize need compute in outside)
// if set uint8, will fuse normalize and quantize to npu
input_attrs[0].type = input_type;
// default fmt is NHWC, npu only support NHWC in zero copy mode
input_attrs[0].fmt = input_layout;

input_mems[0] = rknn_create_mem_from_phys(ctx, input_phys, input_virt,
input_attrs[0].size_with_stride);

...

// Create output tensor memory
rknn_tensor_mem* output_mems[io_num.n_output];
for (uint32_t i = 0; i < io_num.n_output; ++i) {
    output_mems[i] = rknn_create_mem_from_phys(ctx, output_physs[i], output_virts[i],
output_attrs[i].size);
}

// Set input tensor memory
ret = rknn_set_io_mem(ctx, input_mems[0], &input_attrs[0]);
if (ret < 0) {
    printf("rknn_set_io_mem fail! ret=%d\n", ret);
    return -1;
}

// Set output tensor memory
for (uint32_t i = 0; i < io_num.n_output; ++i) {
    // set output memory and attribute
    ret = rknn_set_io_mem(ctx, output_mems[i], &output_attrs[i]);
    if (ret < 0) {
        printf("rknn_set_io_mem fail! ret=%d\n", ret);
        return -1;
    }
}
```

除了引用外部内存的物理地址外，还可以通过引用fd的方式来使用外部分配内存，示例代码如下：

```
int mb_flags = RK_MMZ_ALLOC_TYPE_CMA | RK_MMZ_ALLOC_UNCACHEABLE;

// Allocate weight memory in outside
MB_BLK weight_mb;
rknn_tensor_mem* weight_mem;
ret = RK_MPI_MMZ_Alloc(&weight_mb, mem_size.total_weight_size, mb_flags);
if (ret < 0) {
    printf("RK_MPI_MMZ_Alloc failed, ret: %d\n", ret);
    return ret;
}

void* weight_virt = RK_MPI_MMZ_Handle2VirAddr(weight_mb);
if (weight_virt == NULL) {
```

```
    printf("RK_MPI_MMZ_Handle2VirAddr failed!\n");
    return -1;
}

int weight_fd = RK_MPI_MMZ_Handle2Fd(weight_mb);
if (weight_fd < 0) {
    printf("RK_MPI_MMZ_Handle2Fd failed!\n");
    return -1;
}

weight_mem = rknn_create_mem_from_fd(ctx, weight_fd, weight_virt,
mem_size.total_weight_size, 0);
printf("weight mb info: virt = %p, fd = %d, size: %d\n", weight_virt, weight_fd,
mem_size.total_weight_size);

int mb_flags = RK_MMZ_ALLOC_TYPE_CMA | RK_MMZ_ALLOC_UNCACHEABLE;

// Allocate weight memory in outside
MB_BLK weight_mb;
rknn_tensor_mem* weight_mem;
ret = RK_MPI_MMZ_Alloc(&weight_mb, mem_size.total_weight_size, mb_flags);
if (ret < 0) {
    printf("RK_MPI_MMZ_Alloc failed, ret: %d\n", ret);
    return ret;
}

void* weight_virt = RK_MPI_MMZ_Handle2VirAddr(weight_mb);
if (weight_virt == NULL) {
    printf("RK_MPI_MMZ_Handle2VirAddr failed!\n");
    return -1;
}

int weight_fd = RK_MPI_MMZ_Handle2Fd(weight_mb);
if (weight_fd < 0) {
    printf("RK_MPI_MMZ_Handle2Fd failed!\n");
    return -1;
}

weight_mem = rknn_create_mem_from_fd(ctx, weight_fd, weight_virt,
mem_size.total_weight_size, 0);
printf("weight mb info: virt = %p, fd = %d, size: %d\n", weight_virt, weight_fd,
mem_size.total_weight_size);
```

9.2.2 模型内存的外部分配

在9.1的章节提到模型内存占用有分两部分，一部分是internal内存，另外一部分是weight内存。应用如果需要使用外部分配的模型内存，可以通过接口rknn_set_weight_mem()，rknn_set_internal_mem()接口设置模型weight和internal使用的内存。参考示例如下：

```
// Load RKNN Model
ret = rknn_init(&ctx, model_virt, model_size, RKNN_FLAG_MEM_ALLOC_OUTSIDE, NULL);
TIME_END(rknn_init);
```

```
if (ret < 0) {
    printf("rknn_init fail! ret=%d\n", ret);
    return -1;
}

//query and inset input / output tensor
...

// Allocate weight memory in outside
MB_BLK weight_mb;
rknn_tensor_mem* weight_mem;
ret = RK_MPI_MMZ_Alloc(&weight_mb, SIZE_ALIGN_128(mem_size.total_weight_size), mb_flags);
void* weight_virt = RK_MPI_MMZ_Handle2VirAddr(weight_mb);
int weight_fd = RK_MPI_MMZ_Handle2Fd(weight_mb);
weight_mem = rknn_create_mem_from_fd(ctx, weight_fd, weight_virt,
mem_size.total_weight_size, 0);
ret = rknn_set_weight_mem(ctx, weight_mem);
if (ret < 0) {
    printf("rknn_set_weight_mem fail! ret=%d\n", ret);
    return -1;
}
printf("weight mb info: virt = %p, fd = %d, size: %d\n", weight_virt, weight_fd,
mem_size.total_weight_size);

// Allocate internal memory in outside
MB_BLK internal_mb;
rknn_tensor_mem* internal_mem;
ret = RK_MPI_MMZ_Alloc(&internal_mb, SIZE_ALIGN_128(mem_size.total_internal_size),
mb_flags);
void* internal_virt = RK_MPI_MMZ_Handle2VirAddr(internal_mb);
int internal_fd = RK_MPI_MMZ_Handle2Fd(internal_mb);

internal_mem = rknn_create_mem_from_fd(ctx, internal_fd, internal_virt,
mem_size.total_internal_size, 0);
ret = rknn_set_internal_mem(ctx, internal_mem);
if (ret < 0) {
    printf("rknn_set_internal_mem fail! ret=%d\n", ret);
    return -1;
}
printf("internal mb info: virt = %p, fd = %d, size: %d\n", internal_virt, internal_fd,
mem_size.total_internal_size);
```

9.3 Internal内存复用

RKNN API提供了外部管理NPU内存的机制，通过RKNN_FLAG_MEM_ALLOC_OUTSIDE参数，用户可以指定模型中间的feature内存由外部分配。该功能的典型应用场景如下：

- 部署时，所有NPU内存均是用户自行分配，便于对整个系统内存进行统筹安排。
- 用于多个模型串行运行场景，中间feature内存在不同上下文复用，特别是针对RV1103/RV1106这种内存极为紧张的情况。

例如，下图中有两个模型，模型1的Internal Tensor占用大于模型2，如果模型1和模型2顺序地运行，可以只开辟0x00000000~0x000c4000地址的一块内存给模型1和2共用，模型1推理结束后，这块内存可以被模型2用来读写Internal Tensor数据，从而节省内存。

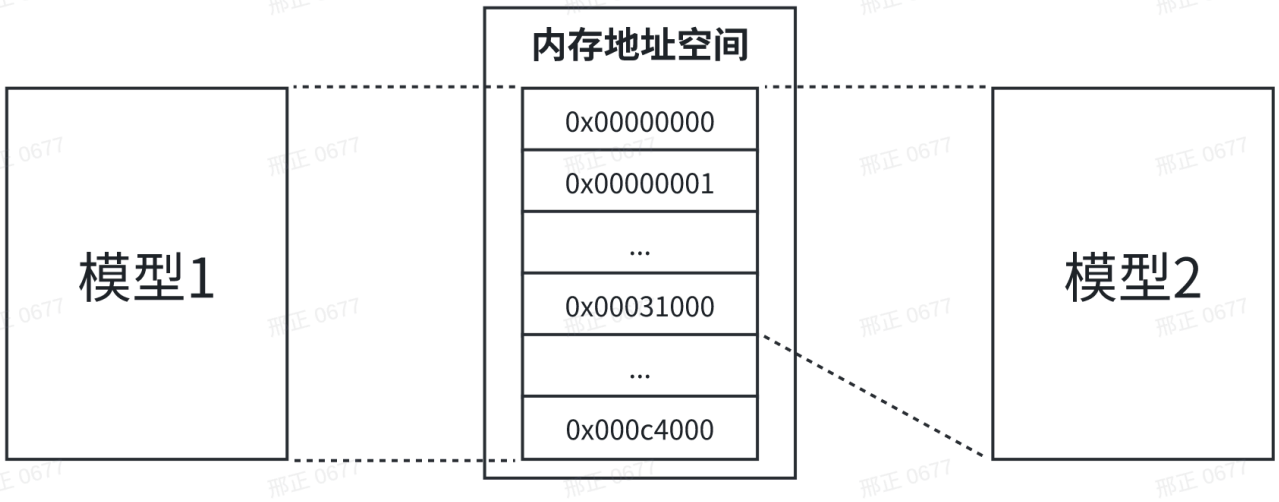


图9-1 两个模型Internal Tensor共享同一块内存地址空间的示例图

假设模型1的路径是model_path_a，模型2路径是model_path_b，示例代码如下：

```
rknn_init(&ctx_a, model_path_a, 0, RKNN_FLAG_MEM_ALLOC_OUTSIDE, NULL);
rknn_query(ctx_a, RKNN_QUERY_MEM_SIZE, &mem_size_a, sizeof(mem_size_a));
rknn_init(&ctx_b, model_path_b, 0, RKNN_FLAG_MEM_ALLOC_OUTSIDE, NULL);
rknn_query(ctx_b, RKNN_QUERY_MEM_SIZE, &mem_size_b, sizeof(mem_size_b));

// 获取两个模型最大的internal size
max_internal_size = MAX(mem_size_a.total_internal_size, mem_size_b.total_internal_size);
internal_mem_max = rknn_create_mem(ctx_a, max_internal_size);

// 设置a模型internal memory
internal_mem_a = rknn_create_mem_from_fd(ctx_a, internal_mem_max->fd,
internal_mem_max->virt_addr, mem_size_a.total_internal_size, 0);
rknn_set_internal_mem(ctx_a, internal_mem_a);

// 设置b模型internal memory
internal_mem_b = rknn_create_mem_from_fd(ctx_b, internal_mem_max->fd,
internal_mem_max->virt_addr, mem_size_b.total_internal_size, 0);
rknn_set_internal_mem(ctx_b, internal_mem_b);
```

9.4 多线程复用上下文

在多线程场景中，一个模型可能会被多个线程同时执行，如果每个线程都单独初始化一个上下文，那么内存消耗会很大，因此可以考虑共享一个上下文，避免数据结构重复构造，减少运行时内存占用。RKNN API提供了复用上下文的接口，接口定义如下：

```
int rknn_dup_context(rknn_context* context_in,rknn_context* context_out)
```

其中，context_in是已初始化的上下文，而context_out是复用context_in的上下文。如下图所示，两个context的模型结构相同，因此可以复用上下文。

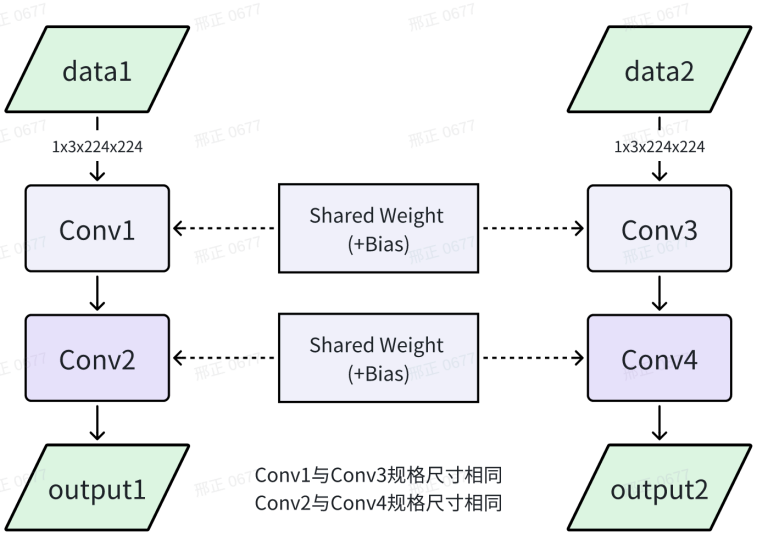


图9-2 两个相同的模型复用上下文的示例图

9.5 多种分辨率模型共享相同权重

当多个不同分辨率模型有相同的权重时，可以共享相同的权重，以减少内存占用。在RKNPU SDK<=1.5.0版本时，此功能可以以较小的内存占用实现不同分辨率模型间的动态切换。在1.5.0版本后，该功能被动态shape功能替代。

如下图所示，模型A和模型B的权重完全相同。

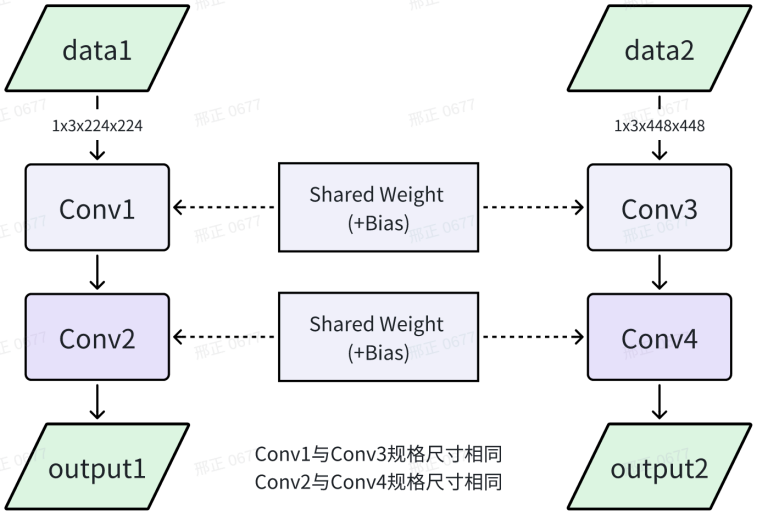


图9-3 两个不同分辨率模型共享权重的示例图

可按照以下步骤实现多分辨率模型共享相同权重：

1. 在转换RKNN模型时，其中一个模型设置为主模型，rknn.config接口设置参数remove_weight=False，另一个模型设置为从模型，设置参数remove_weight=True。主RKNN模型包含权重，从RKNN模型不包含卷积类权重。
2. 部署时，先初始化主RKNN模型，再初始化从RKNN模型。初始化从模型时，使用RKNN_FLAG_SHARE_WEIGHT_MEM标志，并新增rknn_init_extend参数，该参数值为主模型的上下文。假设主模型路径是model_A，从模型路径是model_B，示例代码如下：

```
rknn_context context_A;
rknn_context context_B;
ret = rknn_init(&context_A,model_A,0,0,NULL);

...

rknn_init_extend extend;
extend.ctx = context_A;
ret = rknn_init(&context_B, model_B,0,RKNN_FLAG_SHARE_WEIGHT_MEM,&extend);
```

10 常见问题

10.1 NPU环境准备问题

- **版本兼容性**
  - NPU内核驱动和Runtime版本兼容

    建议将NPU内核驱动升级到0.9.2或之后的版本。在遇到问题时，先更新到最新版本的NPU内核驱动。
  - RKNN-Toolkit2导出的模型和Runtime版本之间的兼容关系如下表所示

表10-1 RKNN模型和Runtime版本对应关系

RKNN模型版本	Runtime版本
1.2.0	>=1.2.0 and <=1.5.0
1.3.0	>=1.3.0 and <=1.5.0
1.4.0	>=1.4.0 and <=1.5.0
1.5.0	1.5.0
1.5.2	>=1.5.2
1.6.0	>=1.5.2
2.0.0	>=2.0.0
2.1.0	>=2.0.0

- **如何更新NPU内核驱动**

  建议升级完整固件以更新NPU驱动，对应固件可以找厂商提供。
- **板端docker环境中如何使用NPU**

  在板端使用docker部署应用时，如果要使用NPU资源，需要在启动容器时，映射NPU相关资源，参考命令如下：

```
docker run -t -i --privileged -v /dev/dri/renderD129:/dev/dri/renderD129 -v
/proc/device-tree/compatible:/proc/device-tree/compatible -v
/usr/lib/librknnrt.so:/usr/lib/librknnrt.so ai_application:v1.0.0 /bin/bash
```

  关注以下参数：
  - **/dev/dri/renderD129**：RK3588 NPU设备节点，Runtime依赖该节点以使能NPU。
  - **/proc/device-tree/compatible**：该文件记录SOC型号，RKNN-Toolkit Lite2等组件依赖该文件获取当前SOC信息。
  - **/usr/lib/librknnrt.so**：Runtime库存放位置，RKNN-Toolkit Lite2和RKNPU2 C API依赖该文件以使用NPU资源。
  - **ai_application:v1.0.0**：待启动容器所使用的镜像名和版本。

10.2 工具安装问题

- **RKNN-Toolkit2依赖的环境限制太严格，导致无法成功安装**

  在所有依赖库都已安装、但部分库的版本和要求不匹配时，可以尝试在安装指令后面加上"no-deps"参数，取消安装Python库时的环境检查。如：

  ```
  pip install rknn-toolkit2*.whl --no-deps
  ```

- **PyTorch依赖说明**

  RKNN-Toolkit2的PyTorch模型加载功能，依赖于PyTorch。PyTorch的模型分为浮点模型和已量化模型（包含QAT及PTQ量化模型）。

  对于PyTorch 1.6.0导出的模型，建议将RKNN-Toolkit2依赖的PyTorch版本降级至1.6.0以免出现加载失败的问题。

  对于已量化模型（QAT、PTQ），我们推荐使用PyTorch 1.10~1.13.1导出模型，并将RKNN-Toolkit2依赖的PyTorch版本升级至1.10~1.13.1。

  另外在加载PyTorch模型时，建议导出原模型的PyTorch版本，要与RKNN-Toolkit2依赖的PyTorch版本尽量一致。

  推荐使用的PyTorch版本为1.6.0、1.9.0、1.10或1.13.1版本。

- **TensorFlow依赖说明**

  RKNN-Toolkit2的TensorFlow模型加载功能依赖于TensorFlow。由于TensorFlow各版本之间的兼容性一般，其他版本可能会造成RKNN-Toolkit2模型加载异常，所以在加载TensorFlow模型时，建议导出原模型的TensorFlow版本，要与RKNN-Toolkit2依赖的TensorFlow版本一致。

  对于TensorFlow版本引发的问题，通常会体现在"rknn.load_tensorflow()"阶段，且出错信息会指向依赖的TensorFlow路径。

  推荐使用的TensorFlow版本为2.6.2或2.8.0。

- **RKNN-Toolkit2安装包命名规则**
  以1.5.2版本的发布件为例，RKNN-Toolkit2 wheel包命令规则如下：

  ```
  rknn_toolkit2-1.5.2+b642f30c-cp38-cp38-linux_x86_64.whl
  ```

  - rknn_toolkit2: 工具名称。
  - 1.5.2: 版本号。
  - b642f30c: 提交号。
  - cp<xx>-cp<xx>: 适用的Python版本，例如cp38-cp38表示适用的Python版本是3.8。
  - linux_x86_64: 系统类型和CPU架构。

  请按照自己所用的操作系统、CPU架构和Python版本安装对应的工具包，否则安装会失败。

- **RKNN-Toolkit2是否有ARM Linux版本**

  RKNN-Toolkit2没有ARM Linux版，如果需要在ARM Linux上使用Python接口进行推理，可以安装RKNN-Toolkit-lite2，该工具可以在ARM Linux上用Python运行推理。

- **bfloat16依赖库安装不上**

  bfloat16的依赖库安装出错，如下：

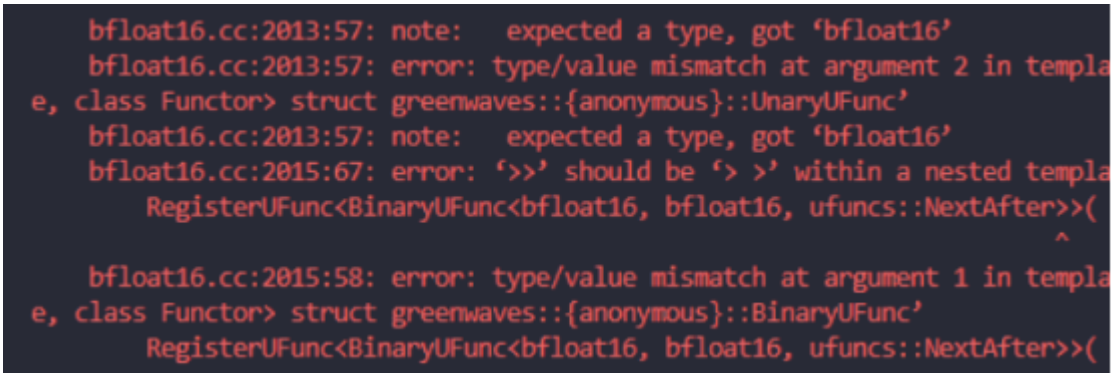


图10-1 bfloat16依赖库安装失败日志

更换pip源为阿里源，或者更新RKNN-Toolkit2至1.5.0以及之后的版本（1.5.0以及之后的版本已经去除bfloat16库的依赖）。

10.3 模型转换常用参数说明

本章节主要覆盖模型转换阶段常用参数的使用说明。

- **根据模型确定参数**

  模型转换时，"rknn.config()"和"rknn.build()"接口会影响模型转换结果。

  "rknn.load_onnx()"，"rknn.load_tensorflow()"指定输入输出节点，会影响模型转换结果。

  "rknn.load_pytorch()"，"rknn.load_tensorflow()"指定输入的尺寸大小会影响模型转换结果。

  可以参考以下基本步骤进行模型转换：

  1. 准备量化数据，提供dataset.txt文件。
  2. 确定模型要使用的NPU平台，如RK3566、RV1106等，并填写rknn.config()接口中的target_platform参数。
  3. 当输入是3通道的图像，且量化数据采用的是图片格式（如jpg、png格式）时，需要确认模型的输入是RGB还是BGR，以决定rknn.config()接口中quant_img_RGB2BGR参数的值。
  4. 确认模型训练时候的归一化参数，以决定rknn.config接口中的mean_values和std_values参数的值。
  5. 确认模型输入的尺寸信息，填入load接口相应参数中，如rknn.load_pytorch()接口中的input_size_list参数。
  6. 确认模型要量化比特数，以决定rknn.config()接口中的quantized_dtype参数的值。**不对模型进行量化或加载的是已量化模型时可以忽略此步骤**。
  7. 确认模型量化时使用的量化算法，以决定rknn.config()接口中quantized_algorithm参数的值。**不对模型进行量化或加载已量化模型时可以忽略此步骤**。
  8. 确认是否对模型进行量化，以决定rknn.build()接口中do_quantization参数的值。选择对模型进行量化时，需要额外填写rknn.build()接口中的dataset参数，指定量化正数据。

- **RKNN模型的跨平台兼容性**

  对于rknn.config()的target_platform设置的平台参数，兼容性关系如下：

  - RK3566、RK3568平台使用的模型是相互兼容的。
  - RK3588、RK3588S平台使用的模型是相互兼容的。
  - RV1103、RV1106平台使用的模型是相互兼容的。

- **量化校正数据的格式及要求**

  量化校正数据的格式有两种选择，一种是图片格式（jpg，png），RKNN-Toolkit2会调用OpenCV接口进行读取；另一种是npy格式，RKNN-Toolkit2会调用numpy接口进行读取。

  对于非RGB/BGR图片输入的模型，建议使用numpy的npy格式提供量化数据。

- **多输入模型dataset.txt文件的填写方式**

  模型量化需要用dataset.txt文件指定量化数据的路径。规则为一行作为一组输入，模型存在多输入时，多个输入写在同一行，并用空格隔开。

  如单输入模型，使用两组量化数据：

  ```
  sampleA.npy
  sampleB.npy
  ```

  如三个输入的模型，两组量化数据按如下方式填写：

  ```
  sampleA_in0.npy sampleA_in1.npy sampleA_in2.npy
  sampleB_in0.npy sampleB_in1.npy sampleB_in2.npy
  ```

- **确认rknn.config()的quant_img_RGB2BGR参数**

  采用图片（jpg，png）作为量化数据时，需要考虑设置quant_img_RGB2BGR参数。

  模型采用RGB图片进行训练时，则quant_img_RGB2BGR参数设为False或不设置。且在使用Python inference接口或RKNPU2 C API进行推理时，输入RGB图片。

  模型采用BGR图片进行训练时，则quant_img_RGB2BGR参数设为True。但在使用Python inference接口或RKNPU2 C API进行推理时，同样需要输入BGR图片（quant_img_RGB2BGR只会影响从量化校正集读入的图像）。

  若量化数据采用numpy的npy格式，则建议不要使用quant_img_RGB2BGR参数，避免产生使用混乱的问题。

- **rknn.config()的mean、std和quant_img_RGB2BGR的计算顺序问题**

  因为quant_img_RGB2BGR只控制在量化过程中读取校正集图像时是否要进行转换通道，并不会影响其他的步骤。因此对于RKNN-Toolkit2的inference接口及 RKNPU2 C API，对输入数据都只先进行减均值（mean）、再除标准差（std）的操作，并没有通道转换的操作。

- **模型是非3通道输入或多输入时，rknn.config()的mean_values和std_values的设置问题**

  mean_values和std_values的设置格式是一致的。以mean_values为例子。

  假设输入有N个通道，则mean_values的值为[[channel_1, channel_2, channel_3, channel_4, ..., channel_n]]。

  存在多输入时，则mean_values的值为[[channel_1, channel_2, channel_3, channel_4, ..., channel_n], [channel_1, channel_2, channel_3, channel_4, ..., channel_n]]。

- **量化参数矫正算法和量化图片数量的选取**

  RKNN-Toolkit2中量化算法（rknn.config()的quantized_algorithm）参数提供三种算法进行参数矫正，分别为normal、mmse和kl_divergence，默认使用normal。normal为常规的量化参数矫正算法；而mmse会迭代中间层的计算结果，对权重数值进行一定范围的裁剪，以获得更高的推理精度。使用mmse不一定能提升量化精度，但相比normal方式，量化时会占用更多的内存、耗费更长的模型转换时间；使用kl_divergence量化算法所用时间会比normal多一些，但比mmse会少很多，在某些场景下（feature分布不均匀时）可以得到较好的改善效果。

  建议先使用normal算法，如果量化效果不佳，可尝试使用mmse或kl_divergence算法。

使用normal或kl_divergence算法时，推荐给出20-200组数据进行量化。使用mmse量化时，推荐使用20-50组数据进行量化。

- **量化模型与非量化模型，推理时输入输出的差异**

  调用通用RKNPU2 C API时（指不使用pass_through、zero_copy的方式调用C API），输入数据的数据类型（如uint8数据，float数据）与模型的量化与否没有关系。输出数据的数据类型可以选择自动处理成float32格式，也可以选择直接输出模型推理结果，此时数据类型与输出节点的数据类型一致。使用Python推理接口会有点差异，具体关系如下表：

表10-2 Python推理接口和通用C API接口区别

模型量化后	Python推理 (rknn.inference())	C API推理(rknn.run()) (非pass_through、zero_copy)
输入类型是否有限制	无限制。 rknn.inference()的输入为numpy数组，本身带有data type属性，该输入会自动转成RKNN模型需要的数据格式。	无限制。 rknn_inputs的rknn_tensor_type参数可以根据实际输入，指定RKNN_TENSOR_FLOAT32、RKNN_TENSOR_FLOAT16、RKNN_TENSOR_INT8、RKNN_TENSOR_UIN8、RKNN_TENSOR_INT16。指定后，会将输入自动转成RKNN模型需要的数据格式。
输出类型是否变化	无变化。 无论模型量化与否，Python的rknn.inference()接口总是返回float类型输出。无法选择其他数据类型。	有变化。 RKNPU2 C API的rknn outputs attr，可以设置want_float=1，得到float类型的输出。而量化后，可以设置want_float=0，此时可以输出最后一个节点的原始输出数据，如i8量化时，输出int8数据。
输入format是否有变化（NCHW，NHWC）	无变化。 无论模型量化与否，rknn.inference()接口的data_format参数，可以根据需要设置为nchw或nhwc。	无变化。 无论模型量化与否，rknn inputs结构体的rknn_tensor_format参数，可以根据需要设置为NCHW或NHWC。

- **是否存在在线预编译的模式**

  RKNN-Toolkit2只支持导出离线预编译的模型，不支持导出在线预编译的模型（RKNN-Toolkit1支持），因此并不存在离线预编译和在线预编译的模式选择。

- **RKNN-Toolkit转出来的RKNN模型可以在RK3566平台上使用吗**

  不可以。

  RKNN-Toolkit转出来的RKNN模型适用于RK1806 / RK1808 / RK3399Pro / RV1109 / RV1126等平台；RK3566平台需要用RKNN-Toolkit2转出来的RKNN模型。RKNN-Toolkit2转出来的RKNN模型适用于RK3566 / RK3568 / RK3588 / RK3588S / RV1103 / RV1106 / RK3562 / RK3576 / RV1126B 等平台。

  RKNN-Toolkit工具的使用说明请参考以下工程：

  https://github.com/airockchip/rknn-toolkit

  RKNN-Toolkit2工具的使用说明请参考以下工程：

  https://github.com/airockchip/rknn-toolkit2

10.4 模型加载问题

10.4.1 RKNN-Toolkit2支持的深度学习框架和对应版本

请参考3.1章节。

10.4.2 各框架的OP支持列表

RKNN-Toolkit2对不同框架的支持程度有差异，详细信息可以参考以下目录中的RKNNToolkit2_OP_Support文档：

https://github.com/airockchip/rknn-toolkit2/blob/master/doc/

10.4.3 ONNX模型转换常见问题

- **加载模型时出现"Error parsing message"报错**

  转换examples/onnx/resnet50v2模型时，提示加载失败：

  ```
  E load_onnx: Catch exception when loading onnx model: /rknn_resnet_demo/resnet50v2.onnx!
  E lod_onnx: Traceback (most recent call last):
  E load_onnx: File "rknn/api/rknn_base.py", line 1094, in
  rknn.api.rknn_base.RKNNBase.load_onnx
  E load_onnx: File "/usr/local/lib/python3.6/dist-packages/onnx/__init__.py", line 115,
  in load_model
  ……
  E load_onnx: google.protobuf.message.DecoderError: Error parsing message
  ```

  原因可能是resnet50v2.onnx模型损坏导致（如没下载全），需要重新下载该模型，并确保其MD5值正确，如：

  ```
  22ed6e6a8fb9192f0980acca0c941414 resnet50v2.onnx
  ```

- **是否支持动态的输入shape**

  1.5.2之前的RKNN-Toolkit2不支持动态的输入shape，比如onnx输入维度为[-1, 3, -1, -1]，表示batch、height和width维度是不固定的。

  1.5.2以及之后的版本可以通过rknn.config()的dynamic_input参数进行动态输入shape的仿真，详见5.4章节。

- **自定义输出节点时报错**

  rknn.load_onnx()时传入outputs参数进行模型的裁剪，但报如下错误：

  ```
  E load_onnx: the '378' in outputs=['378', '439', '500'] is invalied!
  ```

  日志提示输出节点378是无效的，因此outputs参数需设置正确的输出节点名称。

10.4.4 Pytorch模型转换常见问题

- **加载Pytorch模型时出现torch._C没有jit_pass_inline属性的错误**

  错误日志如下：

  ```
  'torch._C' has no attribute '_jit_pass_inline'
  ```

  请将PyTorch升级到1.6.0或之后的版本。

- **Pytorch模型的保存格式**

  目前只支持`torch.jit.trace()`导出的模型。`torch.save()`接口仅保存权重参数字典，缺乏网络结构信息，无法被正常导入并转成RKNN模型。

- **转换时遇到PytorchStreamReader失败的错误**

  详细错误如下：

  ```
  E Catch exception when loading pytorch model: ./mobilenet0.25_Final.pth!
  E Traceback (most recent call last):
  ……
  E cpp_module = torch._C.import_ir_module(cu, f, map_location, extra_files)
  E RuntimeError: [enforce fail at inline container.cc:137]. PytorchStreamReader failed
  reading zip archive: faild finding central directory frame ……
  ```

  出错原因是输入的PyTorch模型没有网络结构信息。

  通常pth只有权重，并没有网络结构信息。对于已保存的模型权重文件，可以通过初始化对应的网络结构，再使用net.load_state_dict()加载pth权重文件。最后通过torch.jit.trace()接口将网络结构和权重参数固化成一个pt文件。得到torch.jit.trace()处理过以后的pt文件，就可以用rknn.load_pytorch()接口将其转为RKNN模型。

- **转换时遇到KeyError的错误**

  错误日志如下：

  ```
  E Traceback (most recent call last):
  ……
  E KeyError: 'aten::softmax'
  ```

  出现形如KeyError: 'aten::xxx'的错误信息时，表示该算子当前版本还不支持。RKNN-Toolkit2在每次版本升级时都会修复此类bug，请使用最新版本的RKNN-Toolkit2试试。

- **转换时遇到"Syntax error in input! LexToken(xxx)"的错误**

  错误日志如下：

  ```
  WARNING: Token 'COMMENT' defined, but not used
  WARNING: There is 1 unused token
  !!!!! Illegal character '"'
  Syntax error in input! LexToken(NAMED_IDENTIFIER, 'fc', 1, 27)
  !!!!! Illegal character '"'
  ```

  该错误的原因有很多种，请按照以下顺序排查：

  1）未继承torch.nn.module创建网络。请继承torch.nn.module基类来创建网络，然后再用torch.jit.trace()生成pt文件。

  2）更新RKNN-Toolkit2 1.4.0或之后的版本，torch建议使用1.6.0，1.9.0，1.10.0或1.13.1版本。

10.4.5 TensorFlow模型转换常见问题

- **Tensorflow1.x模型报错**

  使用rknn.load_tensorflow()接口加载tensorflow1.x模型如出现报错提示：

```
E load_tensorflow: Catch exception when loading tensorflow model:
./yolov3_mobilenetv2.pb!
E load_tensorflow: Traceback (most recent call last):
……
E load_tensorflow: tensorflow.python.framework.errors_impl.InvalidArgumentError: Node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/cond/Switch_1' expects to be colocated
with unknown node 'MobilenetV2/expanded_conv/depthwise/BatchNorm/moving_mean'
E load_tensorflow: During handling of the above exception, another exception occurred:
E load_tensorflow: Traceback (most recent call last):
E load_tensorflow: File "rknn/api/rknn_base.py", line 990, in
rknn.api.rknn_base.RKNNBase.load_tensorflow
……
E load_tensorflow: return func(*args, **kwargs)
E load_tensorflow: File "/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/importer.py", line 431, in import_graph_def
E load_tensorflow: raise ValueError(str(e))
E load_tensorflow: ValueError: Node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/cond/Switch_1' expects to be colocated
with unknown node 'MobilenetV2/expanded_conv/depthwise/BatchNorm/moving_mean'
```

建议：

- 如当前安装的是1.x的TensorFlow，请安装2.x的TensorFlow。
- 更新RKNN-Toolkit2 / RKNPU2至最新版本。

- **TransformGraph类似的报错**

  TensorFlow的模型转成RKNN时报错：

```
Traceback (most recent call last):
File "test.py", line 80, in <module>
input_size_list=[[1, 368, 368, 3]])
File "/usr/local/lib/python3.6/site-packages/rknn/api/rknn.py", line 68, in
load_tensorflow
input_size_list=input_size_list, outputs=outputs)
File "rknn/api/rknn_base.py", line 940, in rknn.api.rknn_base.RKNNBase.load_tensorflow
File "/usr/local/lib/python3.6/dist-
packages/tensorflow/tools/graph_transforms/__init__", line 51, in
TransformGraph.transforms_string, status)
File "/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/errors_impl.py". ;ome 548, in __exit__
C_api.TF_GetCode(self.status.status)
Tensorflow.python.framework.error_impl.InvalidArgumentError: Beta input to batch norm
has bad shape: [24]
```

  原因：

  1）该模型直接调用TensorFlow原生的TransformGraph类进行优化时，也会报上面的错误（RKNN-Toolkit2里同样会调用TransformGraph进行优化，因此也会报同样的错误）。

  2）可能是模型生成时的TensorFlow版本与目前安装的版本已经不兼容了。

  建议：使用1.14.0的TensorFlow版本重新生成该模型，或者寻找其他框架的同类型模型。

- **"Shape must be rank 4 but is rank 0"报错**

加载pb模型时：

```
rknn.load_tensorflow(tf_pb='./model.pb',
                     inputs=["X","Y"],
                     outputs=['generator/xs'],
                     input_size_list=1, INPUT_SIZE, INPUT_SIZE, 3)
```

会产生报错：

```
E load_tensorflow: Catch exception when loading tensorflow model: ./model.pb!
E load_tensorflow: Traceback (most recent call last):
E load_tensorflow: File "/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/importer.py", line 427, in import_graph_def
E load_tensorflow: graph._c_graph, serialized, options) # pylint: disable=protected-
access
E load_tensorflow: tensorflow.python.framework.errors_impl.InvalidArgumentError: Shape
must be rank 4 but is rank 0 for 'generator/conv2d_3/Conv2D' (op: 'Conv2D') with input
shapes: [], [7,7,3,32].
```

原因可能是该模型是多输入模型，rknn.load_tensorflow()的input_size_list没按规范填写，可以参考examples/functions/multi_input_test里的以下用法：

```
rknn.load_tensorflow(tf_pb='./conv_128.pb',
                     inputs=['input1', 'input2', 'input3', 'input4'],
                     outputs=['output'],
                     input_size_list=[[1, 128, 128, 3], [1, 128, 128, 3],
                                      [1, 128, 128, 3], [1, 128, 128, 1]])
```

- **加载模型出错时的排查步骤**

  首先确认原始深度学习框架是否可以加载该模型并进行正确的推理，检查原始模型是否有问题。

  其次请将RKNN-Toolkit2升级到最新版本。如果模型有RKNN-Toolkit2不支持的层（或OP），通过打开调试日志开关，在日志中可以看到哪一个算子是RKNN-Toolkit2不支持的。

  如果仍无法解决，请将使用的RKNN-Toolkit2版本和详细的错误日志反馈给瑞芯微NPU开发团队。

10.5 模型量化问题

- **量化对模型体积的影响**

  分两种情况，当导入的模型是量化的模型时，rknn.build()接口的do_quantization=False会使用该模型里面的量化参数。当导入的模型是浮点的模型时，do_quantization=False不会做量化的操作，但是会把权重从float32转成float16，这块几乎不会有精度损失。这两种情况都减少了模型权重的体积，从而使得整个模型占用空间变小。

- **模型量化时，图片是否需要和模型输入的尺寸一致**

  不需要。RKNN-Toolkit2会自动对这些图片进行缩放处理。但是缩放操作也可能会使图片信息发生改变，对量化精度产生一定影响，所以最好使用尺寸相近的图片。

  如果时非图像格式的校正数据，如npy格式，则需要与模型输入的shape一致。

- **量化校正集是否需要根据rknn_batch_size参数进行修改**

  不需要。

`rknn.build()`的`rknn_batch_size`参数只会修改最后导出的RKNN模型的batch维（由1改为`rknn_batch_size`），并不会影响量化阶段的流程，因此量化校正集还是按照`batch`为1的方式来设置即可。

- **模型量化时，程序运行一段时间后被kill掉或程序卡住**

  在模型量化过程中，RKNN-Toolkit2会申请较多的系统内存，有可能造成程序被kill掉或卡住。

  解决方法：增加电脑内存或增大虚拟内存（交换分区）。

10.6 模型转换问题

- **常见转换bug报错的问题**

  如遇到如下类似转换报错，很可能是由于当前版本存在bug，可尝试将RKNN-Toolkit2更新至最新版本。

  - **infer_shapes类似错误**

```
(op_type:Mul, name:Where_2466_mul): Inferred elem type differs from existing elem type:
(FLOAT) vs (INT64)
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1555, in rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 5409, in
rknn.api.graph_optimizer.GraphOptimizer.run
E build: File "rknn/api/graph_optimizer.py", line 5123, in
rknn.api.graph_optimizer.GraphOptimizer._fuse_ops
E build: File "rknn/api/ir_graph.py", line 180, in rknn.api.ir_graph.IRGraph.rebuild
E build: File "rknn/api/ir_graph.py", line 140, in
rknn.api.ir_graph.IRGraph._clean_model
E build: File "rknn/api/ir_graph.py", line 56, in rknn.api.ir_graph.IRGraph.infer_shapes
E build: File "/home/anaconda3/envs/rk2/lib/python3.6/site-
packages/onnx/shape_inference.py", line 35, in infer_shapes
E build: inferred_model_str = C.infer_shapes(model_str, check_type)
E build: RuntimeError: Inferred elem type differs from existing elem type: (FLOAT) vs
(INT64)
```

或：

```
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1643, in rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 6256, in
rknn.api.graph_optimizer.GraphOptimizer.fuse_ops
E build: File "rknn/api/ir_graph.py", line 285, in rknn.api.ir_graph.IRGraph.rebuild
E build: File "rknn/api/ir_graph.py", line 149, in
rknn.api.ir_graph.IRGraph._clean_model
E build: File "rknn/api/ir_graph.py", line 62, in rknn.api.ir_graph.IRGraph.infer_shapes
E build: File "/usr/local/lib/python3.6/dist-packages/onnx/shape_inference.py", line 35,
in infer_shapes
E build: inferred_model_str = C.infer_shapes(model_str, check_type)
E build: RuntimeError: Inferred shape and existing shape differ in rank: (0) vs (3)
```

或：

```
(op_type:ReduceMax, name:ReduceMax_18): Interred shape and existing shape differ in
rank: (3) vs (0)
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
……
E build: RuntimeError: Interred shape and existing shape differ in rank: (3) vs (0)
```

- **_p_fuse_two_mul类似错误**

```
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1643, in rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 6197, in
rknn.api.graph_optimizer.GraphOptimizer.fuse_ops
E build: File "rknn/api/graph_optimizer.py", line 204, in
rknn.api.graph_optimizer._p_fuse_two_mul
E build: ValueError: non-broadcastable output operand with shape () doesn't match the
broadcast shape (3,2)
```

- **"Segmentation fault"类似错误**

如picodet模型转换报错：

```
I _fold_constant remove nodes = ['Shape_0', 'Gather_4', 'Shape_1', 'Gather_6',
'Unsqueeze_0', 'Concat_8', 'Cast_3']
**Segmentation fault (Core dumped)**
```

- **_p_fuse_mul_into_conv类似错误**

```
    E build: Catch exception when building RKNN model:
    ……
    E build: ValueError: non broadcastable output operand whith shape (1,258,1,256)
doesn`t match the broadcast shape (80256,258,1,256)
```

- **怎么判断算子RKNN是否支持**

  直接进行模型的转换，如果不支持会有相关提示。

  也可参考以下两个算子支持文档：

  1）`RKNN-Toolkit2`发布包的`doc/RKNNToolKit2_OP_Support-x.x.x.md`文档，该文档为各个框架的粗略支持列表

  2）`RKNPU2`发布包的`doc/RKNN_Compiler_Support_Operator_List_vx.x.x.pdf`文档，该文档包含详细的RKNN算子规格支持情况。

- **转换时提示Expand算子不支持**

  建议：

  1）新版本已经支持CPU的Expand，可尝试更新RKNN-Toolkit2 / RKNPU2至最新版本。

  2）修改模型，采用repeat算子来替代expand算子。

- **转换时提示"Meet unsupported dims in reducesum"**

  模型转换出现`Meet unsupported dims in reducesum, dims: 6`，具体如下：

```
D RKNN: [14:54:19.434] >>>>>> start: N4rknn17RKNNInitCastConstE
D RKNN: [14:54:19.434] <<<<<< end: N4rknn17RKNNInitCastConstE
D RKNN: [14:54:19.434] >>>>>> start: N4rknn20RKNNMultiSurfacePassE
D RKNN: [14:54:19.434] <<<<<< end: N4rknn20RKNNMultiSurfacePassE
D RKNN: [14:54:19.434] >>>>>> start: N4rknn14RKNNTilingPassE
D RKNN: [14:54:19.434] <<<<<< end: N4rknn14RKNNTilingPassE
D RKNN: [14:54:19.434] >>>>>> start: N4rknn23RKNNProfileAnalysisPassE
D RKNN: [14:54:19.434] <<<<<< end: N4rknn23RKNNProfileAnalysisPassE
D RKNN: [14:54:19.434] >>>>>> start: OpEmit
E RKNN: [14:54:19.438] Meet unsupported dims in reducesum, dims: 6
Aborted (core dumped)
```

目前RKNN不支持6维的OP，大多数情况下只支持4维。

- **因NonMaxSuppression等后处理Op导致转换报错**
  - `NonMaxSuppression`等后处理Op，RKNN目前不支持。
  - 可以将图的后处理子图部分移除，如：

    ```
    rknn.load_onnx(model='picodet_xxx.onnx', outputs=['concat_4.tmp_0', 'tmp_16'])
    ```

  - 移除的子图在cpu端另行进行处理。
- **"invalid expand shape"类似报错**

  例如`rvm_mobilenetv3_fp32.onnx`转换时出现以下报错：

```
[E:onnxruntime:, sequential_executor.cc:333 Execute] Non-zero status code returned while
running Expand node. Name:'Expand_294' Status Message: invalid expand shape

E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1638, in rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 5529, in
rknn.api.graph_optimizer.GraphOptimizer.fold_constant
E build: File "rknn/api/session.py", line 69, in rknn.api.session.Session.run
E build: File "/home/cx/work/tools/Anaconda3/envs/rknn/lib/python3.8/site-
packages/onnxruntime/capi/onnxruntime_inference_collection.py", line 124, in run
E build: return self._sess.run(output_names, input_feed, run_options)
E build: onnxruntime.capi.onnxruntime_pybind11_state.InvalidArgument: [ONNXRuntimeError]
: 2 : INVALID_ARGUMENT : Non-zero status code returned while running Expand node.
Name:'Expand_294' Status Message: invalid expand shape
```

因为downsample_ratio的输入值会改变模型中间feature的size，所以说这种图本质上是动态图。建议修改模型downsample_ratio的逻辑，不要用输入的数值来控制中间feature的shape。如需使用动态图功能，可在更新1.5.2的RKNN-Toolkit2后，使用动态shape的功能来模拟动态图（同样需要修改模型downsample_ratio的逻辑，不要用输入的数值来控制中间feature的shape，目前动态shape功能只支持输入的shape是可变的情况）。

- **rknn.config()的mean_values报错提示**

  设置`mean/std`为：

```
rknn.config(mean_values=[128, 128, 128], std_values=[128, 128, 128])
```

时转换模型报错：

```
--> Loading model
transpose_input for input_1: shape must be rank 4, ignored
E load_tflite: The len of mean_values ([128, 128, 128]) for input 0 is wrong, expect 32!
```

原因可能是模型的输入不是3通道图像数据（例如输入shape是1x32，非图像数据），此时：

- 需要根据输入通道个数来设置`mean_values` / `std_values`。
- 如果模型不需要指定`mean/std`，`rknn.config()`可以不设置`mean_values` / `std_values`（`mean/std`一般只对图像输入有效）。

- **模型存在4维以上Op时报错（如5维或6维）**

  当模型存在4维以上Op时（如5维或6维），会有如下报错：

```
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1580, in rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/rknn_base.py", line 341, in
rknn.api.rknn_base.RKNNBase._generate_rknn
E build: File "rknn/api/rknn_base.py", line 307, in
rknn.api.rknn_base.RKNNBase._biild_rknn
E build: IndexError: vector::_M_range_check: __n (which is 4) >= this->size() (which is
4)
```

  RKNN目前暂不支持4维以上的OP，可以手工将这些节点去掉。

- **RKNN是否支持动态卷积**

  目前`RK3588/RK3576/RV1126B`平台支持`group`参数为1的动态卷积。其他平台暂不支持。

- **"Not support input data type 'float16'"报错**

  pytorch训练的权重类型为float16的模型，在转换RKNN时出现以下报错：

```
--> Building model
E build: Not support input data type 'float16'
W build: ==================== WARN(3) =================
E rknn-toolkit2 version: 1.3.0-11912b58
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1638, in rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 5524, in
rknn.api.graph_optimizer.GraphOptimzer.fold_constant
E build: File "rknn/api/load_checker.py", line 63, in
rknn.api.load_checker.create_random_data
E build: File "rknn/api/rknn_log.py", line 113, in rknn.api.rknn_log.RKNNLog.e
E build: ValueError: Not support input data type 'float16'!
```

  目前RKNN-Toolkit2还还不支持float16的权重类型的Pytorch模型，需改为float32。

- **动态图相关报错**

  转换模型时，如果出现以下类似报错：

```
E build: ValueError: The Op of 'NonZero' is not support! it will cause the graph to be a
dynamic graph!
```

说明包含该OP的模型为动态图，需要手动修改模型，用其他OP替换或将其移除。

- **RKNN模型大小问题**

  模型转换结束后，可能存在转换出来的RKNN模型比原始模型大的现象，甚至跟模型的输入shape也有关系，这种现象是正常的。因为RKNN模型里不仅仅包含权重和图结构信息，还会有很多NPU的寄存器配置信息，并且为了提高运行效率，可能也会做OP的拆解等操作，这些都会导致RKNN模型变大。

10.7 模拟器推理及连板推理的说明

- **术语说明**

  模拟器推理：`RKNN-Toolkit2`在`Linux x86_64`平台提供模拟器功能，可以在没有开发板的情况下进行模型推理，获取推理结果。（该功能输出结果未必与连板或板端一致，更推荐使用连板推理或板端推理）。

  连板推理：指在开发板已连接电脑的情况下，调用`RKNN-Toolkit2`的`Python API`推理模型，获取推理结果。

  板端推理：指在开发板上调用`RKNPU2`的`C API`接口推理模型，获取推理结果。

- **模拟器推理结果与连板推理结果不一致**

  发生此情况时，可能意味着板端的结果不正确。

  由于硬件和驱动的差异，模拟器不保证可以和板端获取一模一样的结果。但如果差异实在太大，则大概率是板端驱动Bug导致，可以将问题反馈给RK的NPU团队进行分析。

- **连板推理的工作原理**

  使用连板推理时，RKNN-Toolkit2会与板端的RKNN Server进行通信，通信时会将模型、模型的输入由PC端传至板端，随后调用RKNPU2 C API进行模型推理，板端推理完成后将结果回传至PC端。

- **连板推理与板端推理结果有差异**

  连板推理是基于RKNPU2 C API实现的，理论上连板推理结果会与RKNPU2 C API推理结果一致。当这两者出现较大差异时，请确认输入的预处理、数据类型、数据的排布方式（NCHW，NHWC）是否有差异。

  需指出，如差异很小且发生在小数点后3位及之后的数值上，则属于正常现象。差异可能产生在使用不同的库读取图片、转换数据类型等步骤上。

- **板端推理的速度比连板推理更快**

  由于连板推理存在额外的数据拷贝、传输过程，会导致连板推理的性能不如板端的RKNPU2 C API推理性能。因此，NPU实际推理性能以RKNPU2 C API的推理性能为准。

- **涉及连板调试、连板推理功能时，获取详细的错误日志**

  连板推理时，模型的初始化、推理等操作主要在开发板上完成，此时日志信息主要产生在板端上。

  为了获取具体的板端调试信息，可以通过串口进入开发板操作系统。然后执行以下两条命令设置获取日志的环境变量。保持串口窗口不要关闭，再进行连板调试，此时板端的错误信息就会显示在串口窗口上：

```
export RKNN_LOG_LEVEL=5
restart_rknn.sh
```

- **连板推理数据的排布方式说明**

`rknn.inference`在连板推理时输入数据的排布方式可选择为NCHW或NHWC，即`data_format='NCHW'`或`data_format='NHWC'`，默认为NHWC。因为rknn-toolkit2中在转换时onnx模型的4维输入默认都是NCHW的格式，在使用rknn进行inference时输入排布方式默认为NHWC，内部会进行对应的格式转换，如果在inference中设置的data_format='NCHW'，则输入数据不进行默认的NCHW到NHWC的转换。具体例子可参考"https://github.com/airockchip/rknn-toolkit2/issues/220"

10.8 模型评估常见问题

- **量化模型精度不及预期**

  参考本文档的第7章节。

- **支持哪些框架的已量化模型**

  RKNN-Toolkit2 1.4及之后的版本支持TensorFlow、TensorFlow Lite和PyTorch框架的已量化模型。

- **连板调试时，连接设备失败**

  连板精度分析（`rknn.accuracy_analysis()`）时出现如下报错：

```
E accuracy_analysis: Connect to Device Failure (-1)
E accuracy_analysis: Catch exception when init runtime!
E accuracy_analysis: Traceback (most recent call last):
E accuracy_analysis: File "rknn/api/rknn_base.py", line 2001, in
rknn.api.rknn_base.RKNNBase.init_runtime
E accuracy_analysis: File "rknn/api/rknn_runtime.py", line 194, in
rknn.api.rknn_runtime.RKNNRuntime.__init__
E accuracy_analysis: File "rknn/api/rknn_platform.py", line 331, in
rknn.api.rknn_platform.start_ntp_or_adb
```

  或连板推理（`rknn.inference`）时出现如下报错：

```
I target set by user is: rk3568
I Starting ntp or adb, target is RK3568
I Device [0c6a9900ef4871e1] not found in ntb device list.
I Start adb…
I Connect to Device success!
I NPUTransfer: Starting NPU Transfer Client, Transfer version 2.1.0 (b5861e7@2020-11-
23T11:50:36)
D NPUTransfer: Transfer spec = local:transfer_proxy
D NPUTransfer: ERROR: socket read fd = 3, n = -1: Connection reset by peer
D NPUTransfer: Transfer client closed fd = 3

E RKNNAPI: rknn_init, server connect fail! ret = -9(ERROR_PIPE)!
E init_runtime: Catch exception when init_runtime!
E init_runtime: Traceback (most recent call last):
E init_runtime: File "rknn/api/rknn_base.py", line 2001, in
rknn.api.rknn_base.RKNNBase.init_runtime
E init_runtime: File "rknn/api/rknn_runtime.py", line 361, in
rknn.api.rknn_runtime.RKNNRuntime.build_graph
E init_runtime: Exception: RKNN init failed. error code: RKNN_ERR_DEVICE_UNAVAILABLE
```

  原因可能是板端未开启RKNN Server服务，请根据2.2章节相关说明运行板端RKNN Server服务。

- **连板调试时，rknn_init失败，返回-6或模型非法的错误**

错误信息如下：

```
E RKNNAPI: rknn_init, msg_load_ack fail, ack = 1(ACK_FAIL), expect 0(ACK_SUCC)!
D NPUTransfer: Transfer client closed, fd = 4
E init_runtime: Catch exception when init runtime!
E init_runtime: Traceback (most recent call last):
E init_runtime: File "rknn/api/rknn_base.py", line 2011, in
rknn.api.rknn_base.RKNNBase.init_runtime
E init_runtime: File "rknn/api/rknn_runtime.py", line 361, in
rknn.api.rknn_runtime.RKNNRuntime.build_graph
E init_runtime: Exception: RKNN init failed. error code: RKNN_ERR_MODEL_INVALID
```

出现该错误一般有以下几种情况：

- 在生成rknn模型时，不同版本的RKNN-Toolkit2和驱动是有对应关系的，建议将RKNN-Toolkit2 / RKNPU2和开发板的固件都升级到最新的版本。
- 没有正确设置`target_platform`。例如不设置`rknn.config()`接口中的`target_platform`时，生成的RKNN模型只能在RK3566/RK3568上运行。如果要在其他平台上运行（如RK3588/RK3588S/RV1103/RV1106/RK3562），则需要在调用`rknn.config()`接口时设置相应的`target_platform`。
- 如果是在Docker容器中推理时出现该问题，有可能是因为宿主机上的`npu_transfer_proxy`进程没有结束，导致通信异常。可以先退出Docker容器，将宿主机上的`npu_transfer_proxy`进程结束掉，然后再进入容器执行推理脚本。
- 也可能是RKNN模型本身有问题。此时可以用串口连到开发板，在设置环境变量`RKNN_LOG_LEVEL=5`后执行`restart_rknn.sh`，然后重跑程序并将产生的详细日志记录下来，反馈给瑞芯微NPU团队。

- **连板调试时，rknn_init()失败，返回设备不可用的错误**

  错误信息如下：

  ```
  E RKNNAPI: rknn_init, msg_ioctl_ack fail, data_len = 104985, except 102961!
  D NPUTransfer: Transfer client closed, fd = 3
  E init_runtime: Catch exception when init_runtime!
  E init_runtime: Traceback (most recent call last):
  E init_runtime: File "rknn/api/rknn_base.py", line 1961, in
  rknn.api.rknn_base.RKNNBase.init_runtime
  E init_runtime: File "rknn/api/rknn_runtime.py", line 360, in
  rknn.api.rknn_runtime.RKNNRuntime.build_graph
  E init_runtime: Exception: RKNN init failed. error code: RKNN_ERR_DEVICE_UNAVAILABLE
  ```

  该问题的原因比较复杂，请按以下方式排查：

  确保RKNN-Toolkit2 / RKNPU2及开发板的系统固件都已经升级到最新版本。各组件版本查询方法请参考2.2章节。

  另外，需要确保`adb devices`或`rknn.list_devices()`都能看到设备，并且`rknn.init_runtime()`的`target`和`device_id`设置正确。

- **Runtime出现"Invalid RKNN model version 6"报错**

  Runtime上出现以下报错：

```
Loading model …
E RKNN: [09:13:25.728] 6, 1
E RKNN: [09:13:25.728] Invalid RKNN model version 6
E RKNN: [06:28:39.049] rknn_init, load model failed!
Exception: RKNN init failed. error code: RKNN_ERR_FAIL
```

原因：

模型版本与RKNN Runtime不兼容

建议：

参考10.1章节的版本兼容性来升级对应的模型和Runtime版本，或者直接将2者同时升级至最新版本

- **Runtime出现"Invalid RKNN format"报错**

  Runtime上出现以下报错：

```
Loading model …
E RKNN: [06:28:39.048] parseRKNN from buffer: Invalid RKNN format!
E RKNN: [06:28:39.049] rknn_init, load model failed!
rknn_init error ret=-1
```

  原因：

  1）可能是模型转换时的`rknn.config()`的`target_platform`没有设置对，或没有设置（如没有设置默认是RK3566）。

  2）Runtime版本与RKNN-Toolkit2不兼容。

  建议：

  1）设置正确的`target_platform`。

  2）RKNN-Toolkit2与Runtime要一起更新到同一个版本。

- **rknn.inference()耗时与rknn.eval_perf()理论速度不一致**

  因为`rknn.inference()`使用PC + adb的方式进行连板推理，存在着一些固定的数据传输开销，因此与`rknn.eval_perf()`理论速度不一致。

  对于更真实的帧率，建议直接在开发板上使用RKNPU2 C API进行测试。

- **rknn.inference()对多batch的支持**

  RKNN-Toolkit2 1.4.0及之后的版本，可以在构建RKNN模型时就指定输入图片的数量，详细用法参考RKNN-Toolkit2 API手册中关于`rknn.build`接口的说明。

  另外，当`rknn_batch_size`大于1（如等于4时），Python里推理的调用要由：

```
outputs = rknn.inference(inputs=[img])
```

  修改为：

```
img = np.expand_dims(img, 0)
img = np.concatenate((img, img, img, img), axis=0)
outputs = rknn.inference(inputs=[img])
```

  完整示例请参考：`examples/functions/multi_batch/`

- **运行多个RKNN模型**

  运行两个或多个模型时，需要创建多个RKNN对象。一个RKNN对象对应一个模型，类似一个上下文。每个模型在各自的上下文里初始化模型，推理，获取推理结果，互不干涉。这些模型在NPU上推理时是串行进行的。

- **模型推理的耗时非常长，而且得到的结果错误**

  如果推理耗时超过20s，且结果错误，这通常是NPU出现了NPU Hang的BUG。如果遇到该问题，可以尝试更新RKNN-Toolkit2 / RKNPU2到最新版本。

- **模型输入为3维情况下，连板推理结果错误**

  模型的输入为3维情况下，如出现Simulator的仿真结果正确，但连板推理结果错误的情况。原因可能是当前NPU的输入3维支持还不完善，后面会完善3维的支持。

  建议：

  - 先将模型输入改为4维。
  - 更新RKNN-Toolkit2 / RKNPU2至最新版本进行尝试。

- **连板推理结果错误，并且每次都不一致**

  ONNX模型转RKNN后，用Simulator的仿真结果正确，并且每次结果都一致。但在连板推理时结果错误，并且每次都不一致。这种问题可能是板端NPU内核驱动bug导致，此时需要更新板端的NPU内核驱动，并且需要一并更新最新的RKNN-Toolkit2 / RKNPU2。

- **模型存在较多的Resize OP时，出现精度下降问题**

  当ONNX模型里存在较多的Resize OP时，转换为RKNN后出现精度下降。可能的原因是：

  1）精度下降是因为NPU目前还不支持硬件级别`Resize`（后续会支持），转换工具会将`Resize`转为`ConvTranspose`，会导致一点点的精度丢失。

  2）如模型有多个串联的`Resize`，则可能会累积了太多误差导致精度下降比较多。

  建议：

  1）目前尽量避免`Resize`的使用（如将`Resize`改为`ConvTranspose`再进行训练）

  2）可以在`rknn.config()`里加入`optimization_level=2`的参数，此时`Resize Op`会走CPU，精度不会掉，但会导致性能下降。

- **do_quantization设为False以后推理结果都为nan**

  `rknn.build()`接口中的`do_quantization`设为`True`时推理结果没有异常，但设为`False`以后推理结果就都变为`nan`了。原因可能是`do_quantization=False`时，RKNN模型的运算类型是fp16的，但该模型的中间层（如卷积）输出的范围可能超出了fp16（65536）的范围（如-51597~75642）。

  建议：

  训练的时候需要保证中间层的输出不超过fp16的表达范围（一般通过添加BN层来解决该问题）。

- **QAT模型与RKNN模型结果不一致**

  在Pytroch框架下使用QAT训练了一个分类模型并转为RKNN模型，对该模型使用Pytorch和RKNN分别进行推理，发现得到的结果不一样，原因可能是Pytorch的推理没有设置`engine='qnnpack'`，因为RKNN的推理方式与`qnnpack`更为贴近。

- **怎么获取模型运行时候内存占用率**

  可以使用`rknn.eval_memory()`接口，输出的日志里有个Total项，就是总的占用大小。

- **性能评估时，开启或关闭rknn.init_runtime()的perf_debug参数，性能数据的差异**

开启`perf_debug`时，为了收集每层的信息，会添加一些调试代码，并且可能禁用一些并行的机制，因此耗时比`perf_debug=False`时多一些。

开启`perf_debug`的主要作用是看模型中是否有耗时占比比较多的层，以此为依据来设计优化方案。

- **环境用的docker，之前连板推理正常，重启docker后，推理时卡在初始化环境阶段?**

  因为docker重启时`npu_transfer_proxy`类似于异常退出的状态，导致开发板上的RKNN Server无法检测到上端连接已经断开，这时需要重启下开发板，重置RKNN Server的连接状态。

10.9 C API使用常见问题

- **rknn_outputs_release()是否会释放rknn_output数组**

  `rknn_outputs_release()`与`rknn_outputs_get()`配合调用，它只释放`rknn_output`数组里的buf。类似的情况还有`rknn_destroy_mem()`。

- **rknn_create_mem如何创建合适的大小的内存?**

  对于输入而言，一般原则是：如果是量化RKNN模型，`rknn_create_mem()`使用`rknn_tensor_attr`的`size_with_stride`分配内存；非量化模型`rknn_create_mem()`使用用户填充的数据类型的`字节数` * `n_elems`分配内存。

  对于输出而言，`rknn_create_mem()`使用用户填充的数据类型的`字节数` * `n_elems`分配内存。

- **输入数据如何填充?**

  如果使用通用API，对于四维形状输入，`fmt=NHWC`，即数据填充顺序为`[batch, height, width, channel]`。非四维输入形状，`fmt=UNDEFINED`，按照模型的原始形状填充数据。

  如果使用零拷贝API，对于四维形状输入，`fmt=NHWC/NC1HWC2`。`fmt=NC1HWC2`时如何填充数据请参考《RKNN Runtime零拷贝调用》章节。非四维输入形状，`fmt=UNDEFINED`，按照模型的原始形状填充数据。

- **pass_through如何使用?**

  输入数据格式由`rknn_query()`的`RKNN_NATIVE_INPUT_ATTR`命令获取。如果是4维形状：对于通道值为1、3、4，layout要求使用`NHWC`，其他通道值要求使用`NC1HWC2`；如果是非4维形状，建议指定`layout=UNDEFINED`。

  另外，在`pass_through`模式下，量化模型通常指定输入Tensor的`dtype`为`INT8`，非量化模型通常指定输入Tensor的`dtype`为`FLOAT16`。

- **出现"failed to submit"错误如何处理?**

  如果错误出现在第一层卷积并且使用零拷贝接口，可能的原因是输入tensor内存分配不够导致，此时应该使用tensor属性中的`size_with_stride`分配内存。

  如果错误出现在中间的NPU层，可能的原因是模型配置出错，此时可在错误日志中找到最新的SDK网盘链接，建议升级最新工具链或者在转换RKNN模型时将该层指定到CPU上运行。

- **出现"Meet unsupport xxx operator"错误如何处理?**

  在板端运行demo出现类似的报错时，一般是板端的`Runtime（librknnrt.so）`不支持该算子。建议用户先更新RKNN 相关工具链到最新版本，再重新转换模型，并在板端重跑demo。

  如果最新的工具链还出现同样报错，则用户需要自行添加该算子的实现，可以参考5.5 章节来自定义实现算子，或者通过redmine上报给RKNN 团队。

- **动态shape模型是否支持在零拷贝流程中使用外部分配的内存?**

  1.6.0之前版本不支持，1.6.0版本开始支持使用`RKNN_FLAG_MEM_ALLOC_OUTSIDE`标志初始化上下文。

- **自定义算子性能评估(runtime)**

自定义算子的性能可以在板端上设定`RKNN_LOG_LEVE=4`或`5`以上, 并运行测试demo, runtime库就会自动打印出每层耗时信息, 包含自定义OP的耗时。

注意: 这层自定义OP的耗时包含compute回调函数在内的所有耗时, 以GPU OP为例子, 耗时会包含compute回调函数在内, 包括`clImportMemoryARM`函数的, 以及数据格式和类型的转换的耗时(包含给输入输出的转换)和最后GPU OP的kernel运行耗时。

11 相关资源

RKNN：https://github.com/airockchip/rknn-toolkit2。

Model Zoo：https://github.com/airockchip/rknn_model_zoo。

RGA：https://github.com/airockchip/librga。